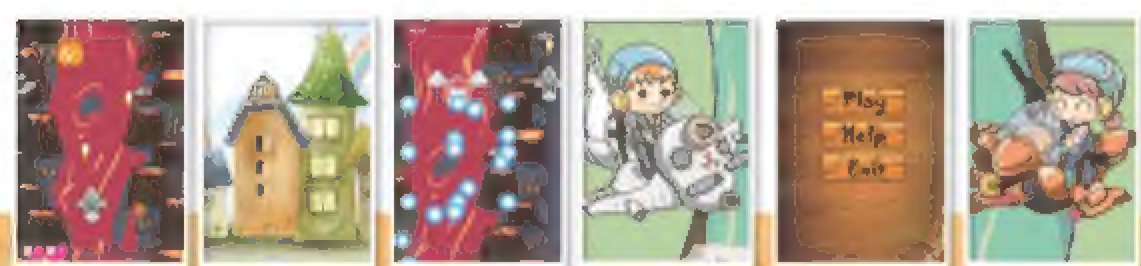
Android 游戏编程之从零开始

Android game programming from scratch





李华明　编著



CD光盘：本书各章范例代码

清华大学出版社



Android

游戏编程之从零开始

李华明　编著

清华大学出版社
北　　京

内 容 简 介

本书主要系统地讲解了 Android 游戏开发，从最基础部分开始，让零基础的 Android 初学者也能快速学习和掌握 Android 游戏开发。

本书一共 8 章，内容包括 Android 平台介绍与环境搭建、Hello，Android！项目剖析、游戏开发中常用的系统组件、游戏开发基础、游戏开发实战、游戏开发提高篇、Box2d 物理引擎、物理游戏实战。随书光盘包括全书 65 个项目源代码。

本书适合 Android 游戏开发的初学者使用，也适合作为 Android 游戏培训的教材和高校游戏专业师生的参考书。

图书在版编目（CIP）数据

Android 游戏编程之从零开始 / 李华明编著. — 北京：清华大学出版社，2011.10

ISBN 978-7-302-26535-1

I. ①A… II. ①李… III. ①移动电话机—游戏程序—程序设计 IV. ①TN929.53②TP317

中国版本图书馆 CIP 数据核字（2011）第 173271 号

责任编辑： 夏非彼　马颖君
责任校对： 闫秀华
责任印制： 杨　艳
出版发行： 清华大学出版社　　**地　址：** 北京清华大学学研大厦 A 座
http://www.tup.com.cn　　**邮　编：** 100084
社 总 机： 010-62770175　　**邮　购：** 010-62786544
投稿与读者服务： 010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈： 010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者： 北京艺辉印刷有限公司
经　销： 全国新华书店
开　本： 190×260　　**印　张：** 25.25　　**字　数：** 646 千字
附光盘 1 张
版　次： 2011 年 10 月第 1 版　　**印　次：** 2011 年 10 月第 1 次印刷
印　数： 1~4000
定　价： 59.00

产品编号：042413-01

前言

如今的Android系统市场份额已节节攀升，势不可挡，越来越多的开发者加入到Android应用开发的行列。从2010年的数据表明，Android系统仅仅推出两年已超过诺基亚的Symbian系统，而且2010年Android市场应用也相比2009年增长了6倍之多；最值得一提的是，这些与日俱增的Android应用程序中，无论是按使用量还是总收入排名，70%的应用排行榜首都是游戏。

本书以Java语言为主系统讲解了Android游戏开发，从最基础的内容开始，让读者循序渐进地学习和掌握Android游戏开发的知识与技巧。对于有Java基础的读者，能够更容易、更快地掌握，当然，阅读本书不需要读者有移动设备的开发经验。

本书总共8章，每章都以前一章的知识点作为铺垫展开，所以对于刚接触Android游戏开发的读者，建议从前往后依次逐章学习。各章知识点整体以从易到难、从浅到深的形式呈现，所以建议读者在阅读本书时一定不要跳读，否则学习起来可能会事倍功半。本书各章讲解的内容如下：

第1章介绍Android平台的趋势与发展，以及Android应用开发环境的搭建。

第2章通过一个最简单的Android项目代码对Android开发的基础概念进行详细讲解。

第3章介绍游戏开发中常用的一些基础控件以及布局等。

第4章介绍Android游戏开发的方法，讲解了在Android平台进行游戏开发的一些常用框架、游戏开发的基础概念以及游戏开发相关类的说明。

第5章介绍“飞行射击”游戏的开发，本章是对前几章内容的一个综合演练，尤其对第4章各模块知识点的综合运用，通过本章的学习读者将熟悉和掌握游戏开发流程。

第6章是游戏开发提高部分，主要介绍Android系统手机的一些特性与独有功能，蓝牙对战游戏开发、网络手机通信也都将在本章进行讲解。

第7章讲解在Android系统中结合Box2D物理引擎进行游戏开发的方法。

第8章讲解“迷宫小球”和“堆房子”两个Box2d物理游戏的实战开发。

本书中讲解的知识点基本与Android SDK版本无关，也就是说开发出的应用在Android操作系统的任意版本下都可以运行，没有版本之间的限制。当然也有一些内容只有在SDK较高版本才会有的功能，但是都会在书中有详细的标注与提示，比如有关蓝牙功能的开发需要用到Android 2.0版本。

在本书的撰写过程中，有幸得到游戏源手机游戏研发技术总监桂志刚及其教学团队的大

力支持。他们从实际研发及一线教学实践出发，立足学员需求和未来职业发展，为本书的定位、知识体系及应用实例的选择提供了诸多宝贵建议。本书课程和教学体系在其机构进行了实践应用，取得了较为理想的效果。在此，诚挚感谢游戏源游戏开发培训机构为本书提供实践应用的平台。

在此，我要特别感谢我的家人，完成本书编写的动力主要就是来自家人对我关心与支持。同时也要感谢清华大学出版社图格事业部的夏毓彦老师对本书的出版做了大量的工作，他的 Email 是 booksaga@163.com。

由于编者水平有限，书中难免有疏漏之处，望广大读者指正批评，意见与建议请 Email 至 xiaominghimi@vip.qq.com。也可以在编者的博客上交流：http://blog.csdn.net/xiaominghimi

编者

2011 年 7 月



光盘使用说明



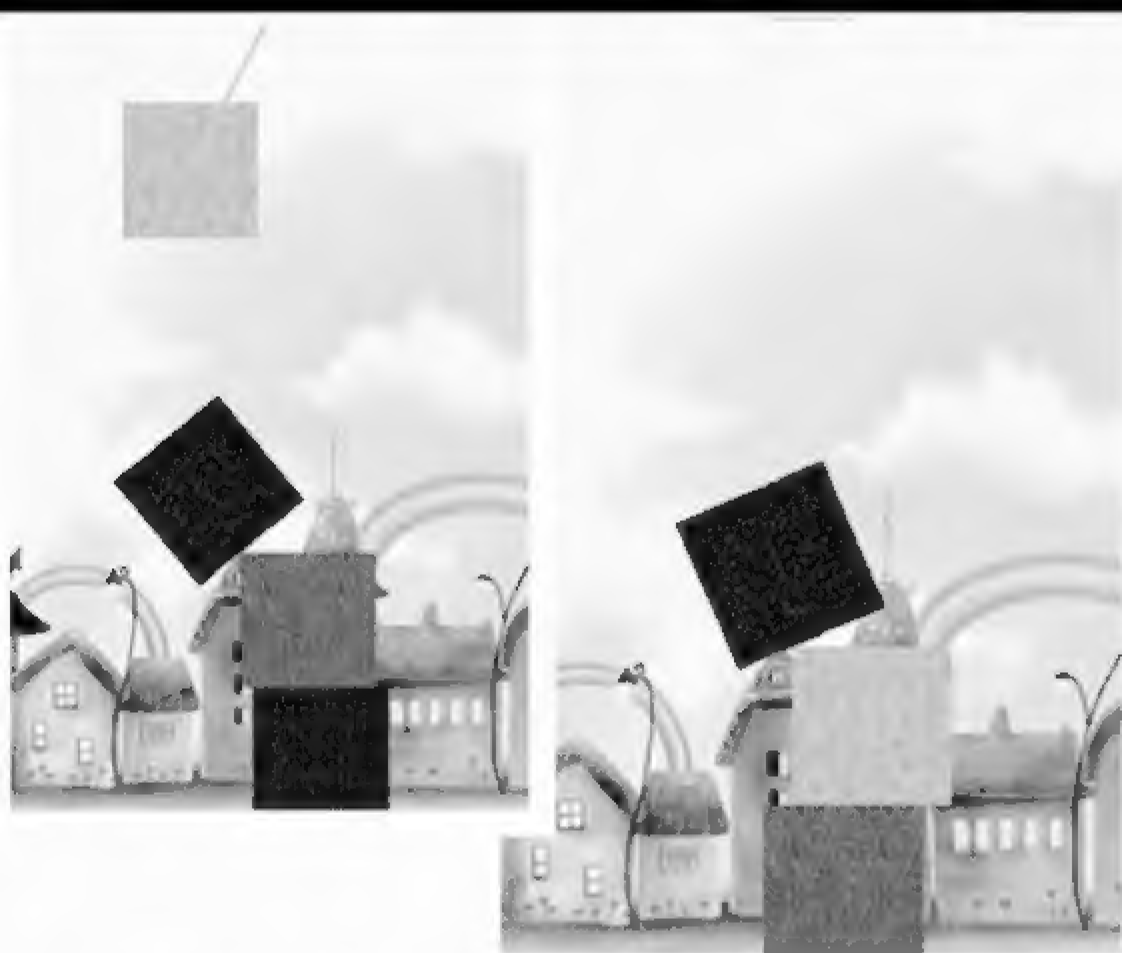
光盘包括以下Android游戏项目代码:

2-1(Activity生命周期)
3-1(Button与点击监听器)
3-10-1(列表之ArrayAdapter适配)
3-10-2(列表之SimpleAdapter适配)
3-11(Dialog对话框)
3-12-5(Activity跳转与操作)
3-12-6(横竖屏切换处理)
3-3(ImageButton图片按钮)
3-4(EditText文本编辑)
3-5(CheckBox与监听)
3-6(RadioButton与监听)
3-7(ProgressBar进度条)
3-8(SeekBar 拖动条)
3-9(Tab分页式菜单)
4-10(可视区域)
4-11-1(Animation动画)
4-11-2-1(动态位图)
4-11-2-2(帧动画)
4-11-2-3(剪切图动画)
4-13(操作游戏主角)
4-14-1(矩形碰撞)
4-14-2(圆形碰撞)
4-14-4(多矩形碰撞)
4-14-5(Region碰撞检测)
4-15-1 (MediaPlayer音乐)
4-15-2(SoundPool音效)
4-16-1(游戏保存之SharedPreference)
4-16-2(游戏保存之Stream)
4-3(View游戏框架)
4-4(SurfaceView游戏框架)
4-7-1(贝塞尔曲线)
4-7-2(Canvas画布)
4-8(Paint画笔)
4-9(Bitmap位图渲染与操作)
5-1(飞行射击游戏实战)
6-1(360° 平滑游戏摇杆)
6-10-1(Socket协议)
6-10-2(Http协议)
6-11(本地化与国际化)
6-2(多触点缩放位图)
6-3(触屏手势识别)
6-4(加速度传感器)
6-5(9patch工具)
6-6(截屏)
6-8(游戏视图与系统组件)
6-9(蓝牙对战游戏)
7-10-1(遍历Body)
7-10-2(Body的m_userData)
7-11(为Body施加力)
7-12(Body碰撞监听)
7-13-1(距离关节)
7-13-2(旋转关节)
7-13-3(齿轮关节)
7-13-4(滑轮关节)
7-13-5-1(通过移动关节移动Body)
7-13-5-2(通过移动关节绑定两个Body动作)
7-13-6(鼠标关节-拖拽Body)
7-14(AABB获取Body)
7-16-1(迷宫小球)
7-4(Box2d物理世界)
7-5(在物理世界中添加矩形)
7-7(添加自定义多边形)
7-9(在物理世界中添加圆形)
8-1(迷宫小球)
8-2(堆房子)




Play
Help
Exit
Little Palace & Maze


Android
游戏编程之从零开始
Back
Game By Himi
Game Win
Replay
Menu















目录 CONTENTS

第 1 章　Android 平台介绍与环境搭建

第 2 章　Hello，Android！

第 3 章 Android 游戏开发常用的系统控件

第 4 章 游戏开发基础

第 5 章 游戏开发实战演练

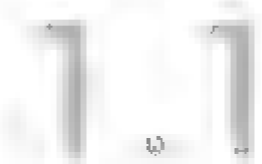
第 1 章

Android 平台介绍与环境搭建

从本章节可以学习到：

- ❖ Android 平台简介
- ❖ Android 开发环境的搭建





Android 平台简介

Android 一词的本义指“机器人”，也是 Google 公司用来作为 2007 年 11 月 5 日宣布的基于 Linux 平台的开源手机操作系统的名称，该平台由操作系统、中间件、用户界面和应用软件组成，是首个为移动终端打造的真正开放和完整的操作系统。Google 公司分别在 2009 年 4 月 28 日发布了 Android 1.5 SDK，2009 年 9 月 16 日发布了 Android 1.6 SDK，2010 年 1 月 5 日发布了 Android 2.1。目前最新版本是 2010 年 12 月 6 日发布的 Android 2.3 Gingerbread 和 2011 年 2 月 3 日发布的、专用于平板电脑的 Android 3.0 Honeycomb 操作系统。

1.1.1　Android 操作系统平台的优势和趋势

说到 Android 操作系统平台的优势，不得不提到最突出的两个特点“免费”和“开源”。

- 免费：Android 免费提供其操作系统，让移动电话制造商可以免费搭载 Android 操作系统，使得手机的制造成本大大降低，渐渐使得 Android 普及。
- 开源：Android 手机操作系统源代码的开放性，不仅让开发者可以在统一开放平台进行程序开发，而且可以解决现今市场不同智能机之间因文件格式不同造成信息交流不便、程序内容无法移植等问题；并且 Android 的开源就意味着手机使用者不必再被动地接受移动电话制造商默认的设置和环境，使用者完全可以根据自己的需求和想法自定义手机的配置。

2010 年数据表明，Android 系统推出 2 年时间已经超越了诺基亚（Nokia）的 Symbian 系统，而且 Android 市场应用数量也相比去年增加了 6 倍之多。这里值得一提的是，这些与日俱增的 Android 应用程序中，无论是按使用量还是总收入来排名，70%的应用排行榜首都是游戏。

如今 Android 的发展趋势势不可挡，Android 已经成为移动设备开发行业中不得不学的平台之一。有关 Android 平台的介绍，这里只是简单地进行概述，如果大家想详细了解的话，可以参考其他书籍或者在网上自行查阅相关知识。

1.1.2　Android SDK 与 Android NDK

Android SDK（Software Development Kit）是 Android 软件开发工具包，用于辅助 Android 操作系统软件开发，是开发 Android 软件、文档、范例、工具的一个集合。

Android NDK（Native Development Kit）类似于 Android SDK，Android 操作系统刚发布的时候，限定所有的应用程序开发都使用 Java 语言进行编写，后来为方便 C/C++开发者更快

地进入 Android 开发行列以及让开发者更直接地接触 Android 系统资源，就推出了 NDK，使得利用传统的 C/C++语言也可以编写 Android 程序。

本书使用 Java 语言和 Android SDK 进行讲解。

Android 开发人员的网站网址为：developer.android.com，上面可以查阅到 Android SDK、开发指南、API 说明等信息。读者在以后的开发中可以不断地查阅这些相关内容。

1.2 Android 开发环境的搭建

本节只向大家介绍在 Windows 下配置 Android 开发环境，有关 Linux、Mac OS 等系统下配置 Android 开发环境的方法请自行查阅相关书籍。

1.2.1 搭配环境前的准备工作

Android 开发环境搭建前，需要下载 Java JDK、Eclipse 、Android SDK 以及 ADT。

- JDK：是整个 Java 的核心，包括了 Java 的运行环境（Java Runtime Envirnment）、类库以及 Java 开发工具等等。
- Eclipse：简单而言就是一个 IDE 集成开发环境。
- Android SDK：Android 开发工具包，内含 Android 虚拟设备，即 Android 模拟器。
- ADT：是 Google 研发的一个插件，此插件集成在 Eclipse 中，可为开发 Android 提供专属开发环境，并且 ADT 中包括创建实例、运行和除错等功能。

1. Java JDK 下载

Java JDK 的下载地址为：http://www.oracle.com/technetwork/java/javase/downloads/index.html。

因为 Sun 公司被 Oracle 公司收购了，所以 JDK 需要从 Oracle 公司网站下载。在如图 1-1 所示的页面上单击“Java”图标，进入下一页面，如图 1-2 所示。

在如图 1-2 所示的界面中：

步骤1 根据提示选择当前使用的电脑操作系统；

步骤2 选中同意许可协议检查框；

步骤3 单击“Continue”（继续）按钮进入下一页面，如图 1-3 所示。

在图 1-3 所示的下载界面中，单击“jdk-6u24-windows-i586.exe”链接进行下载。

2. Android SDK 的下载

Android SDK 的下载地址:http://dl.google.com/android/archives/android-sdk-windows-1.6_r1.zip。按此链接下载下来的 SDK 包含 1.5 和 1.6 版本。

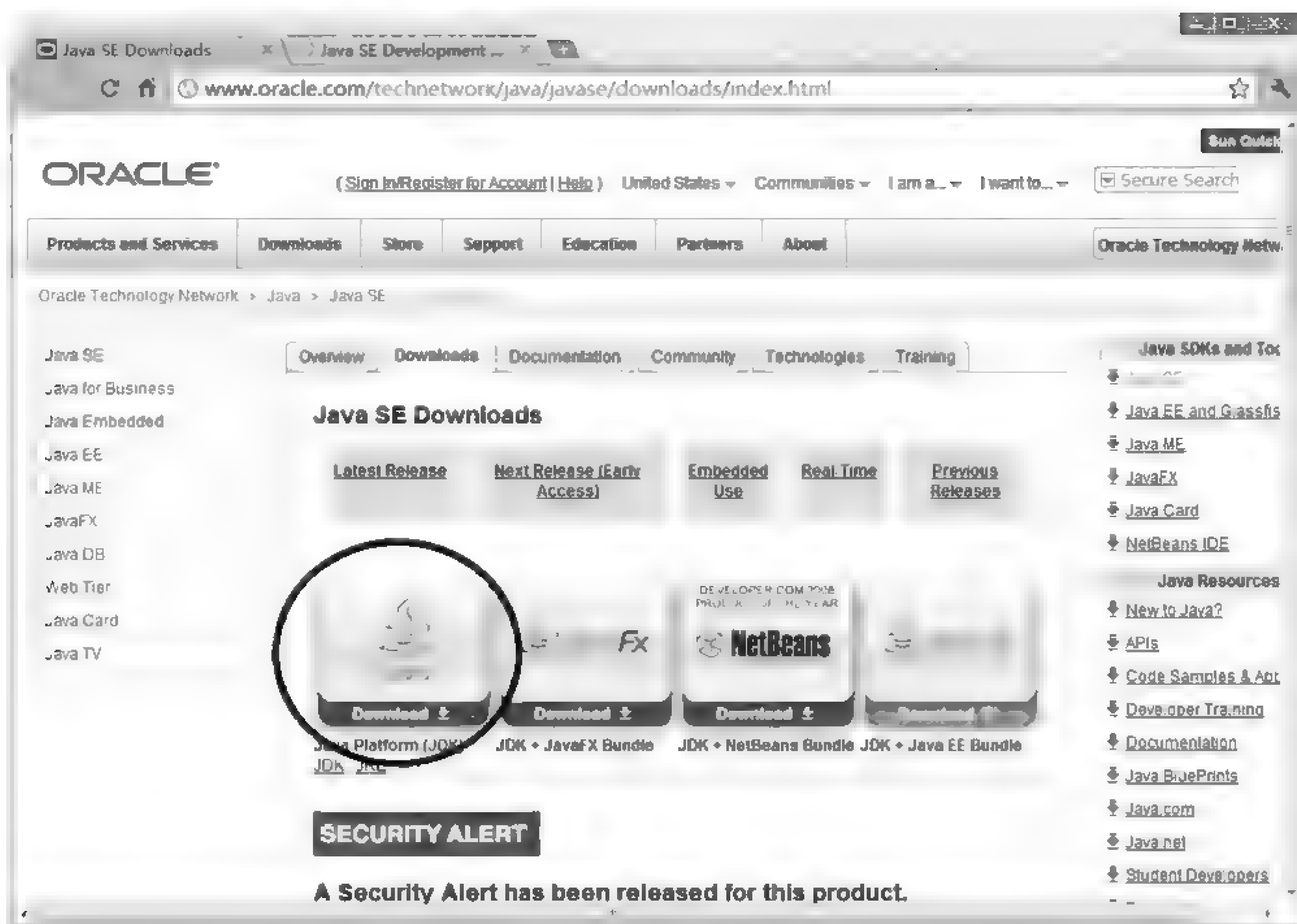


图 1-1　JDK 选择界面

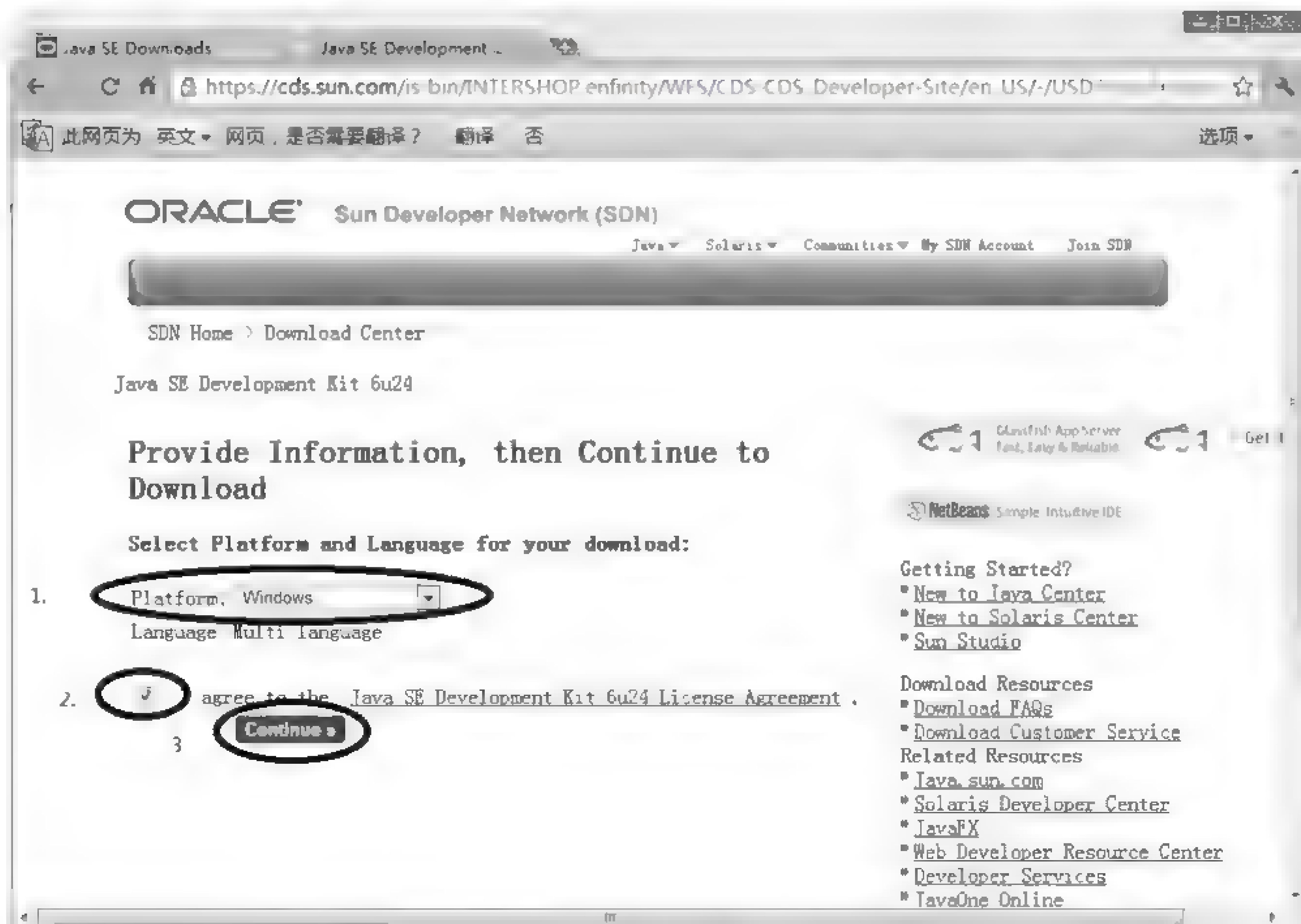


图 1-2　选择 JDK 版本页面

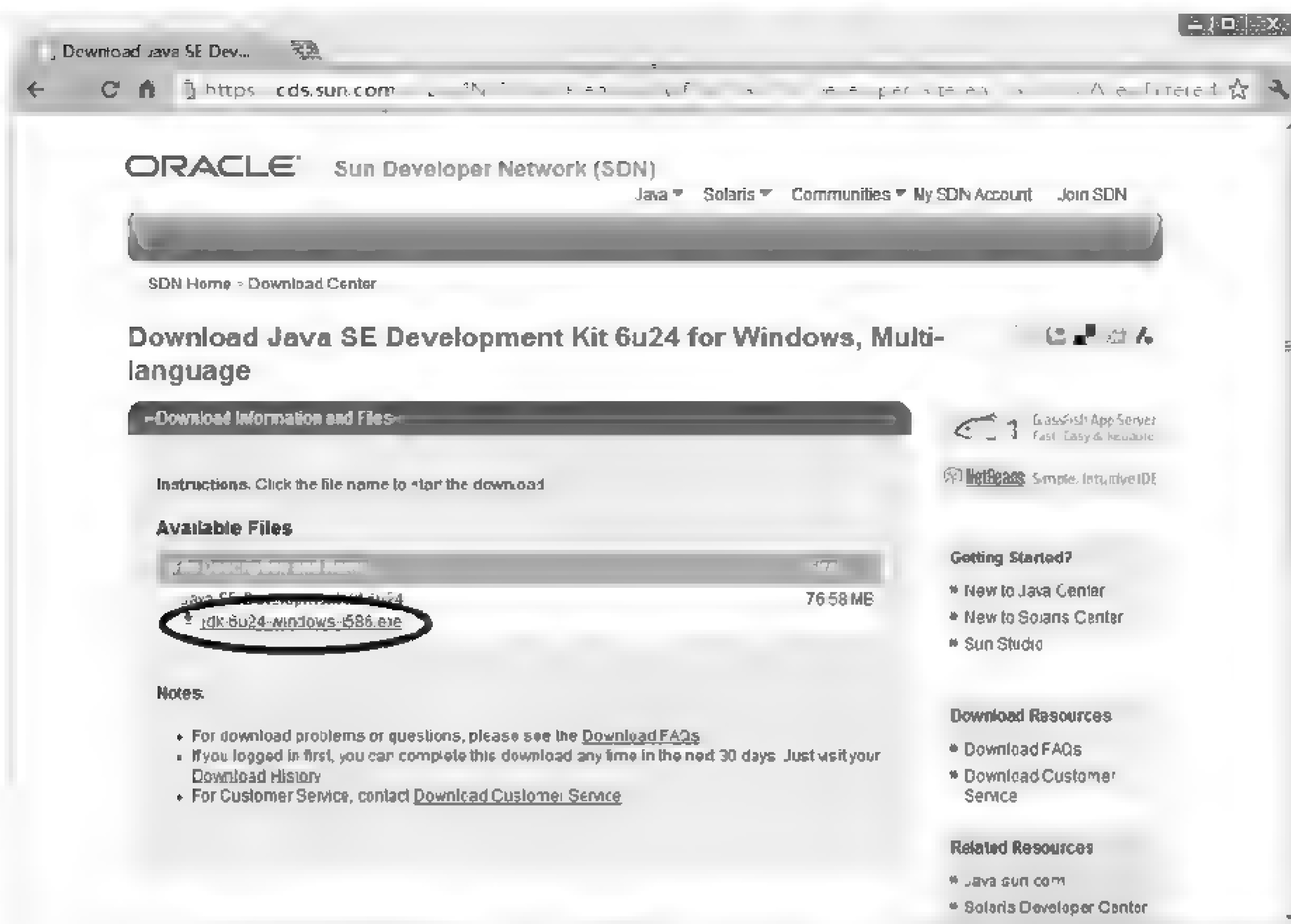


图 1-3 JDK 下载界面

3. Eclipse IDE 的下载

Eclipse IDE 的下载地址为 http://www.eclipse.org/downloads/。从浏览器中输入下载地址，打开如图 1-4 所示的页面，选中“Eclipse Classic 3.6.1”经典版本进行下载。

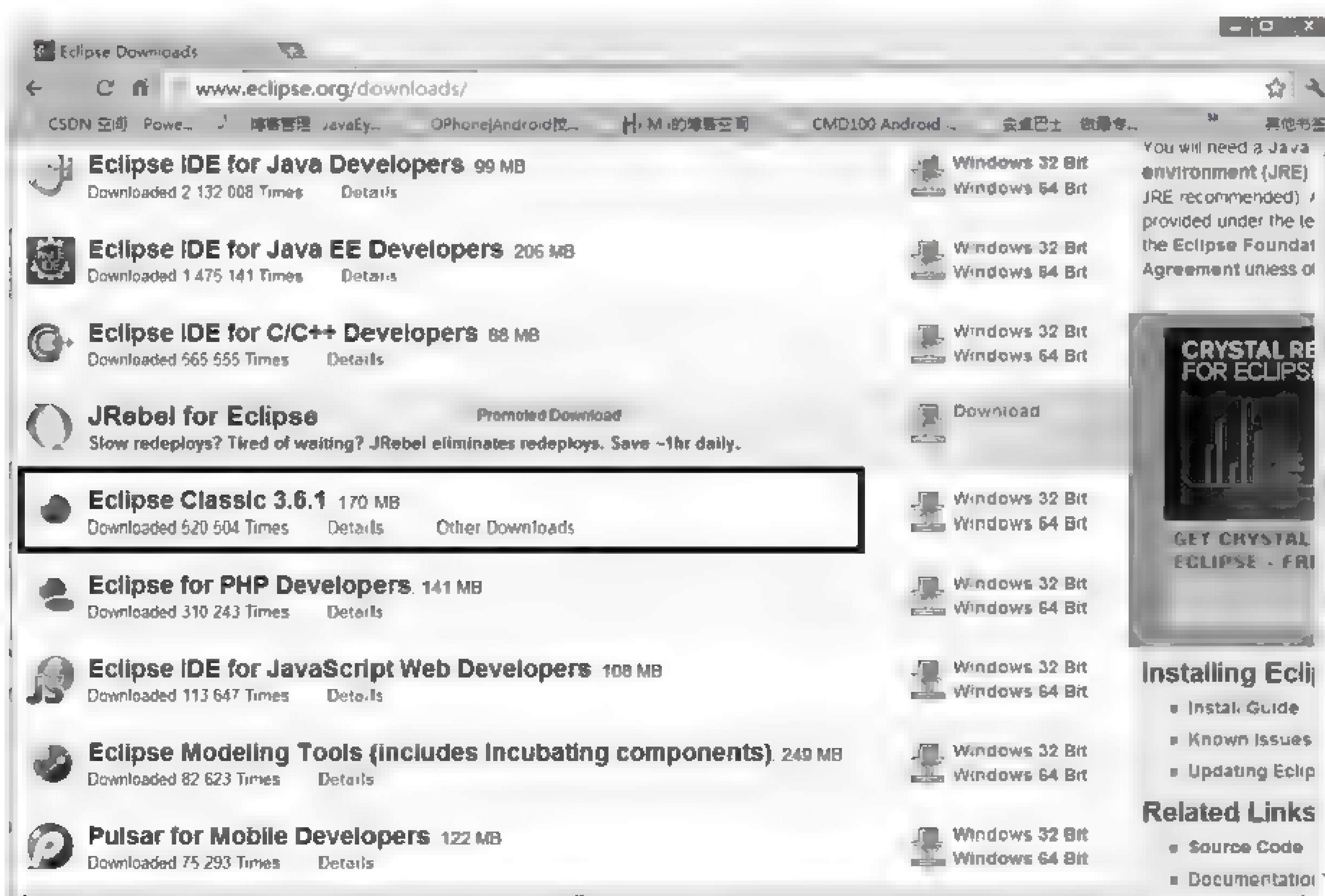


图 1-4 Eclipse 官方网站下载网页

这里需要说明的是：使用 Android 开发应用程序时，不仅需要 Eclipse 主程序还需要 Eclipse JDT（Java Development Tools）和 WTP（Web Tools Platform）这两个开发工具插件。由于 Eclipse Classic 3.4 以上的版本已经包含了这两个插件，所以在配置 Android 开发环境的时候，只要选取 Eclipse Classic 3.4 以上的版本，这样就不用额外地安装 JDT 和 WTP 这两个插件了。单击下载 Eclipse 时，根据电脑的操作系统来选择其是 32 位还是 64 位的版本。

1.2.2 安装和配置环境

Eclipse 运行环境的前提是电脑已经安装 JDK 才可以打开，所以解压 Eclipse 之前需要先安装 JDK，在开发工具的安装过程中，其安装路径最好是英文，尽可能避免中文路径，以免出现不可预知的 BUG。环境具体安装和配置的步骤如下：

步骤1 安装 JDK，安装的路径随意。

步骤2 然后解压 Eclipse，解压路径随意。

步骤3 在 Eclipse 解压目录下找到“eclipse.exe”文件，然后单击启动 Eclipse，可以使用右键在电脑桌面上发送一个快捷方式以方便启动。启动 Eclipse 的过程中会弹出窗口，如图 1-5 所示。

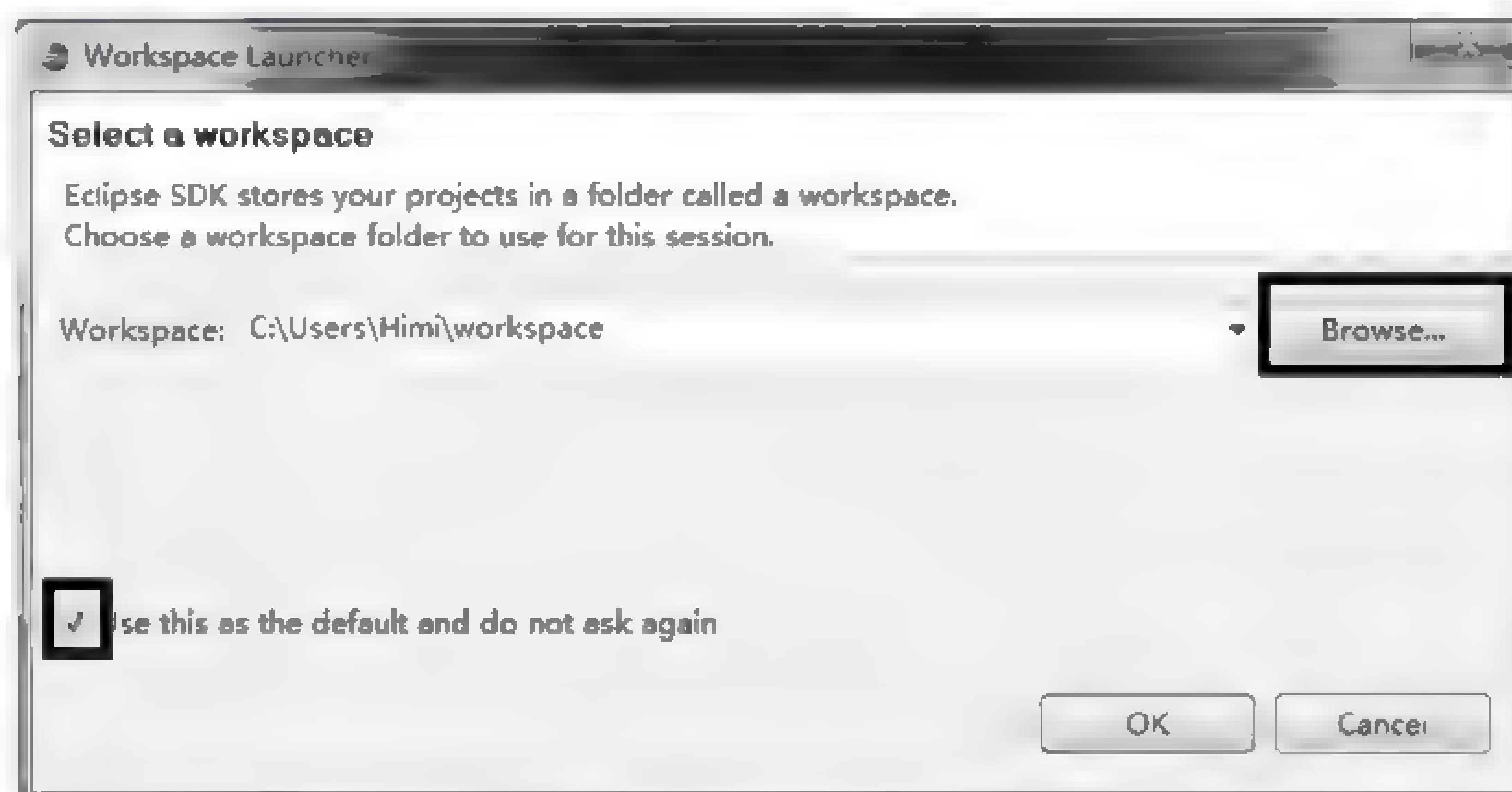


图 1-5 Eclipse 预配置

这里单击“Browse”按钮选择 Eclipse 的工作路径，如果下次启动 Eclipse 不想再显示此窗口，可以选中此界面的左下角，表示下次默认使用此路径（关于 Eclipse 常用的快捷键或者操作大家可以阅读相关资料，本书中会适当讲解一些）。

步骤4 接下来开始安装 ADT。进入 Eclipse 主界面，然后打开“Help”菜单，选中“Install New Software”选项，如图 1-6 所示。

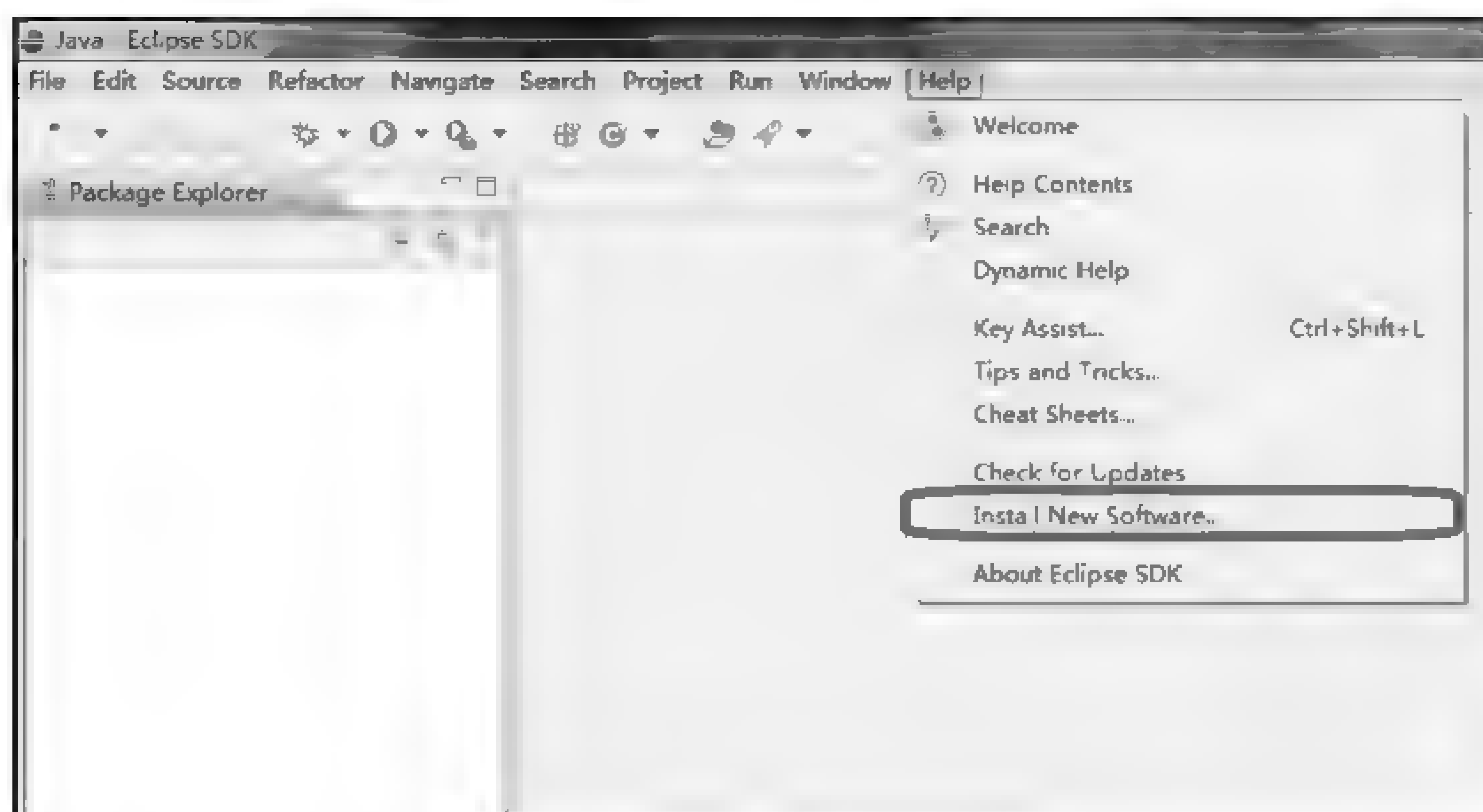


图 1-6 Eclipse 选择安装插件

ADT 的安装方式有两种，在线下载安装和离线安装，分别说明如下。

(1) 在线下载安装

进入安装插件的界面，单击“Add”按钮，在“Location”文本框中填入以下网址：http://dl-ssl.google.com /android/eclipse/，如图 1-7 所示。



图 1-7 Android ADT 选择插件路径界面

然后将“Developer Tools”全部选中，单击“Next”按钮，ADT 安装完成之后，Eclipse 会提示重启 Eclipse，同意使之重启即可完成 ADT 的安装。

（2）离线安装

先自行下载 Android ADT 的离线安装包，这里给出 Android ADT 0.9.7 版本的下载地址：http://dl.google.com/android/ADT-0.9.7.zip，然后在图 1-7 所示的选择插件路径界面中，单击“Archive”按钮，找到下载好的 ADT 插件路径。后续操作如同在线安装的步骤一样。如果处于断网的状态，在单击“Next”按钮之前，建议大家将左下角的“Contact all updata...”这一选项的选中取消，无需检查更新，从而加快安装速度。

这里要注意一点，当离线安装的时候，使用“Eclipse Classic”的版本很难正常离线安装，这是因为仍然缺少必要的插件，解决方法是 Eclipse 使用“Eclipse IDE for java EE”版本即可。

步骤5 配置 Android SDK。

（1）解压下载好的 Android SDK，然后单击 Eclipse 主菜单“Window”下的“Perferences”选项，进入配置，如图 1-8 所示。

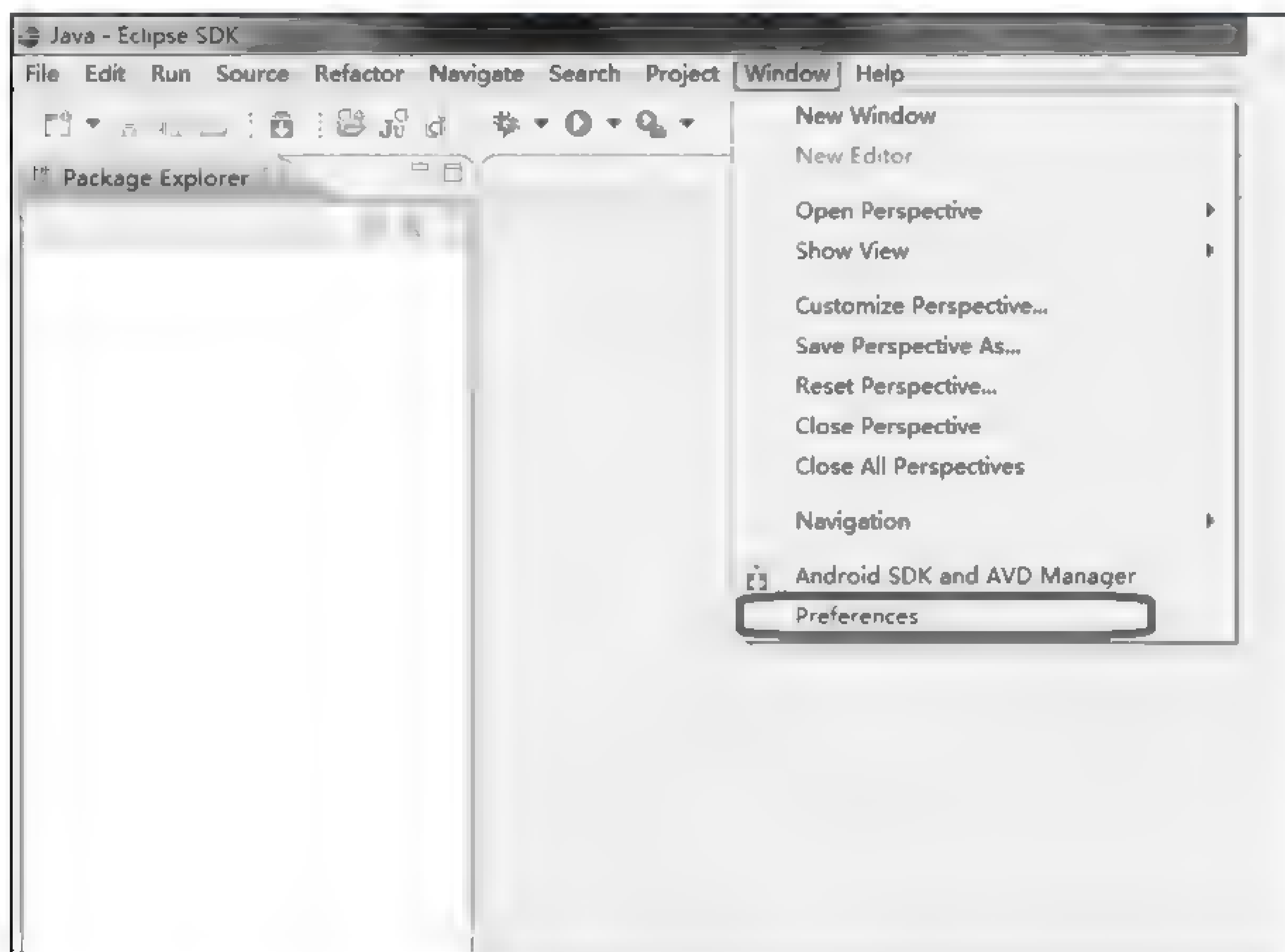


图 1-8 Eclipse 配置选择

（2）如果 ADT 正确安装了，那么在“Preferences”界面的左侧会出现“Android”一栏，单击“Android”选项，此时右侧单击“Browse...”按钮，选择 Android SDK 解压后的路径，如图 1-9 所示。

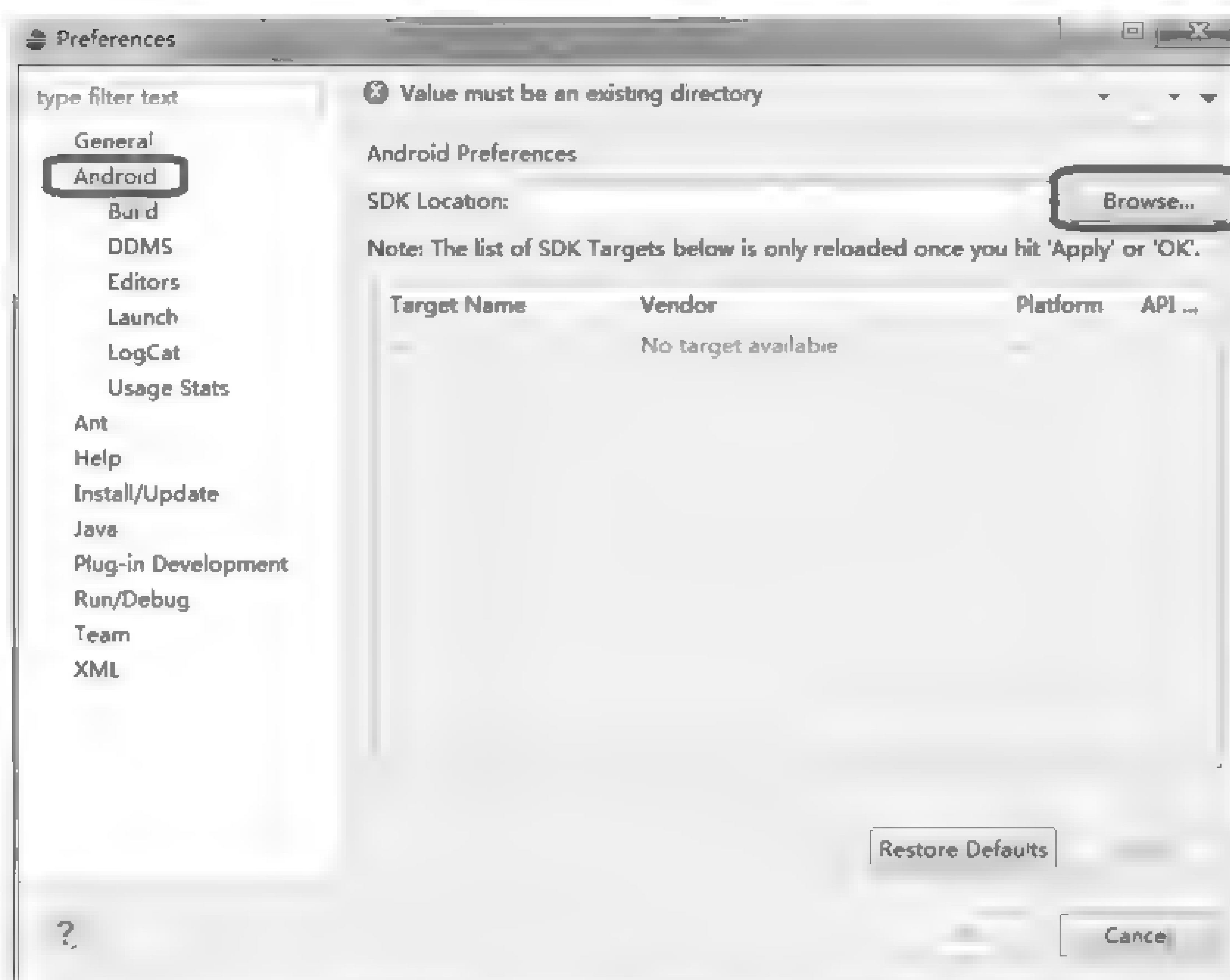


图 1-9　Eclipse 配置选择

路径选择正确之后，在下方会显示当前 SDK 中包含的版本，然后单击“OK”按钮，完成 SDK 配置，如图 1-10 所示。

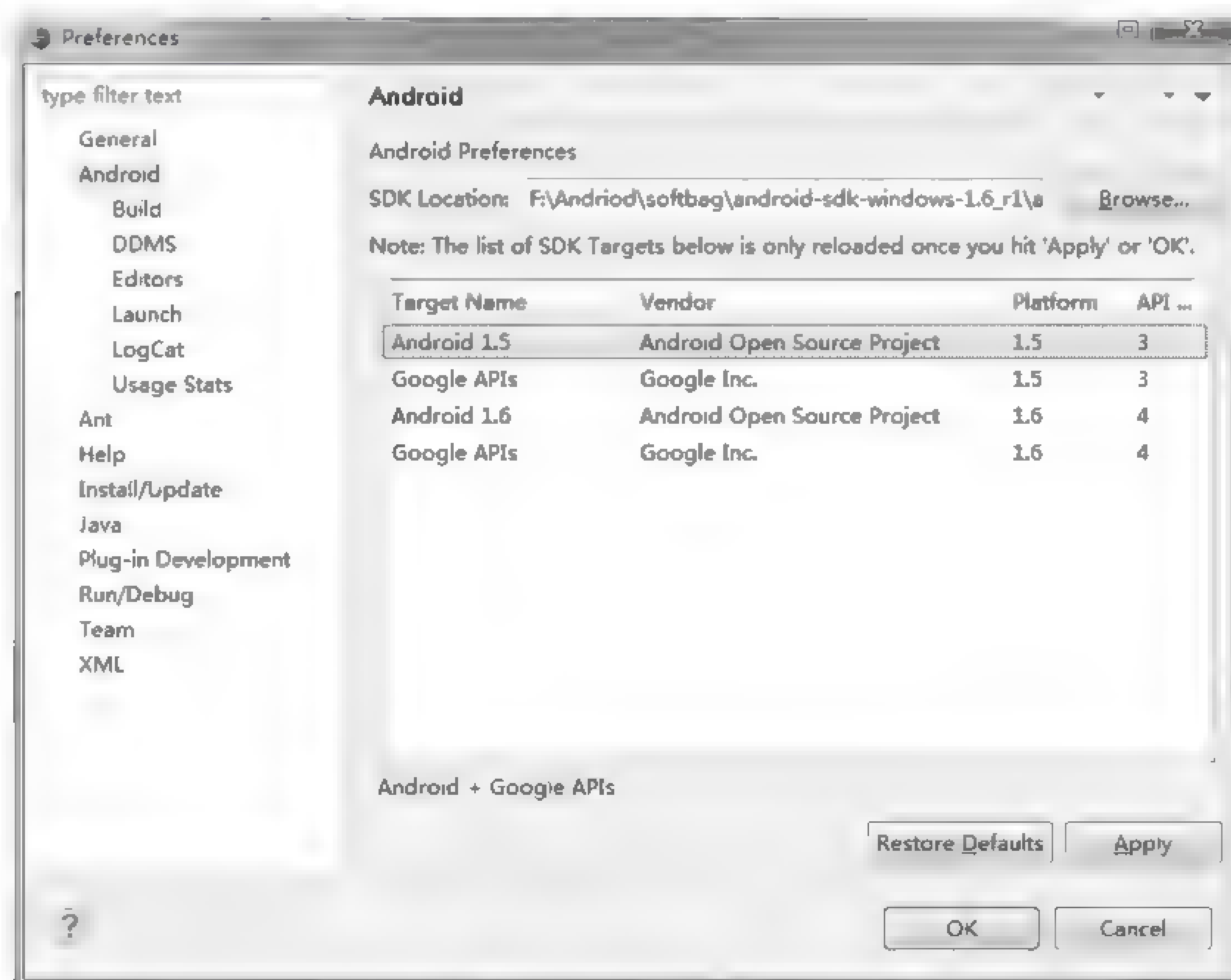


图 1-10　Eclipse 配置选择

步骤6 创建 AVD（Android 模拟器）

AVD 是 Android Virtual Device 的缩写，指 Android 运行的虚拟设备。创建 AVD 有两种方式，第一种通过 Eclipse IDE 来进行创建，另一种则使用命令行进行创建。下面来介绍利用 Eclipse 开发环境创建 AVD 的步骤：

当 SDK 路径配置正确之后，在 Eclipse 的主界面的左上角可以看到一个机器人的标志，如图 1-11 所示，单击进入 SDK&AVD 管理界面，如图 1-12 所示。

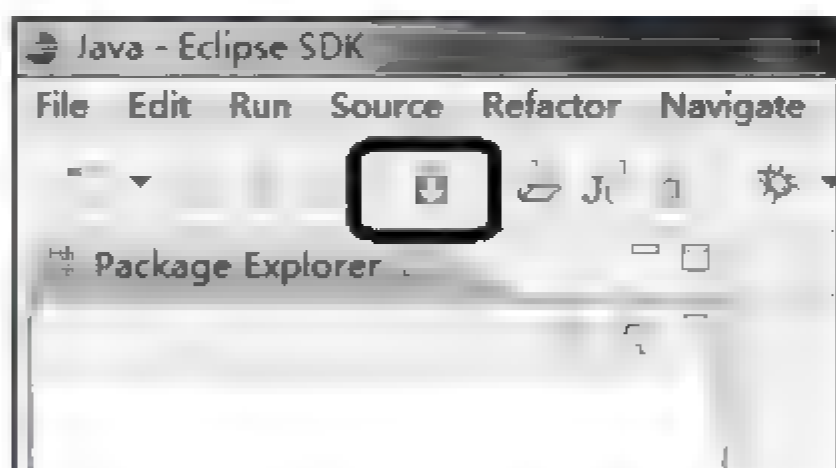


图 1-11 选择 AVD 标志

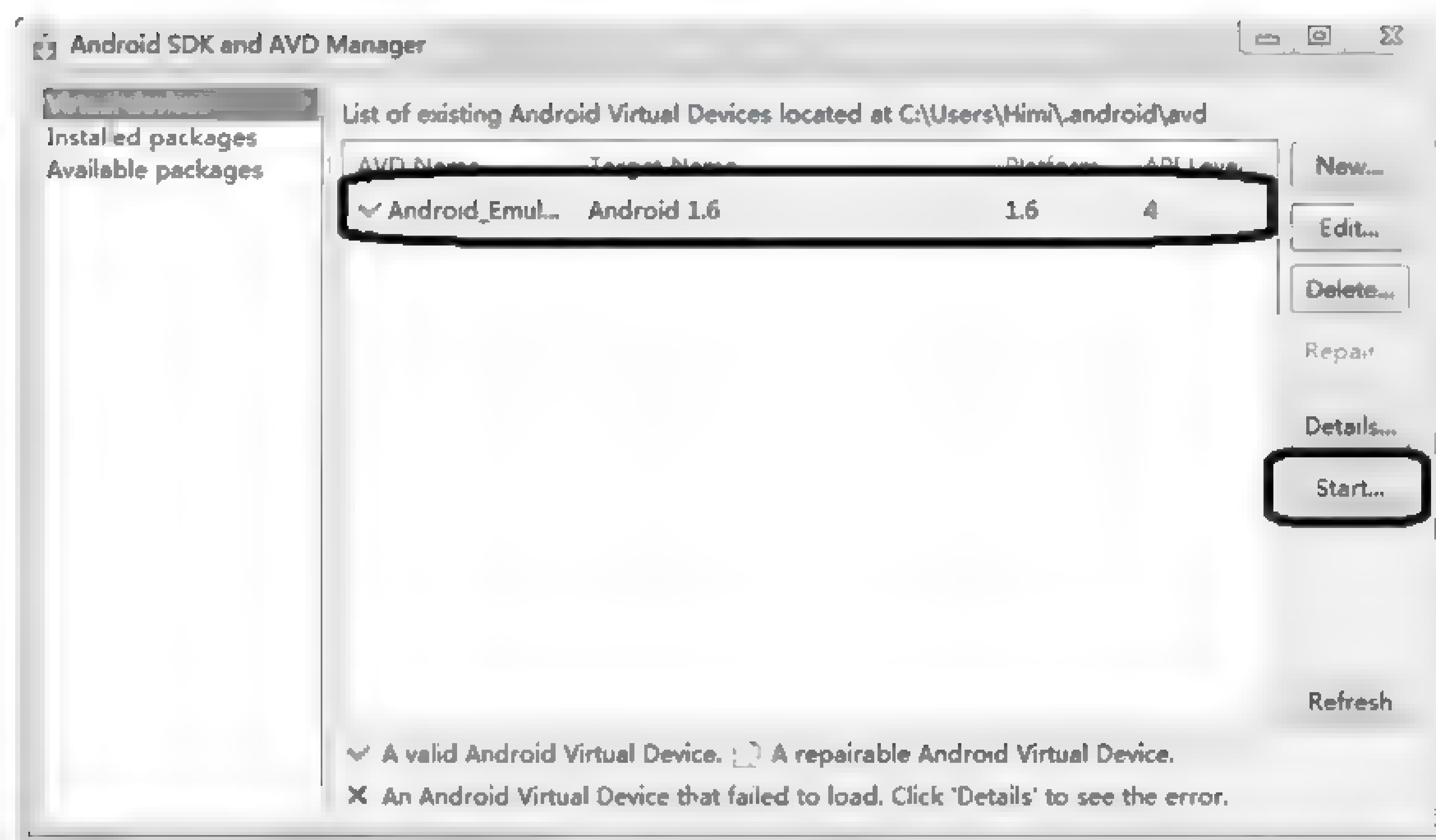


图 1-12 SDK&AVD 管理界面

在 SDK&AVD 管理界面中可以进行添加、删除、编辑 AVD 以及对 SDK 版本更新等操作。这里先讲解如何添加一个新的 AVD。

单击管理界面中的“New”选项，然后在弹出的“Create new Android Virtual Device”界面中，如图 1-13 所示，来手动设置 Android 模拟器的属性，例如：模拟器的名称、分辨率、SDK 版本、是否创建 SDcard 等等。这里先来创建一个简单 AVD，在图 1-13 所示的界面中，只是给模拟器起个名字以及选择一个 SDK 版本，最后单击下方的“Create AVD”按钮即可完成 AVD 的创建。

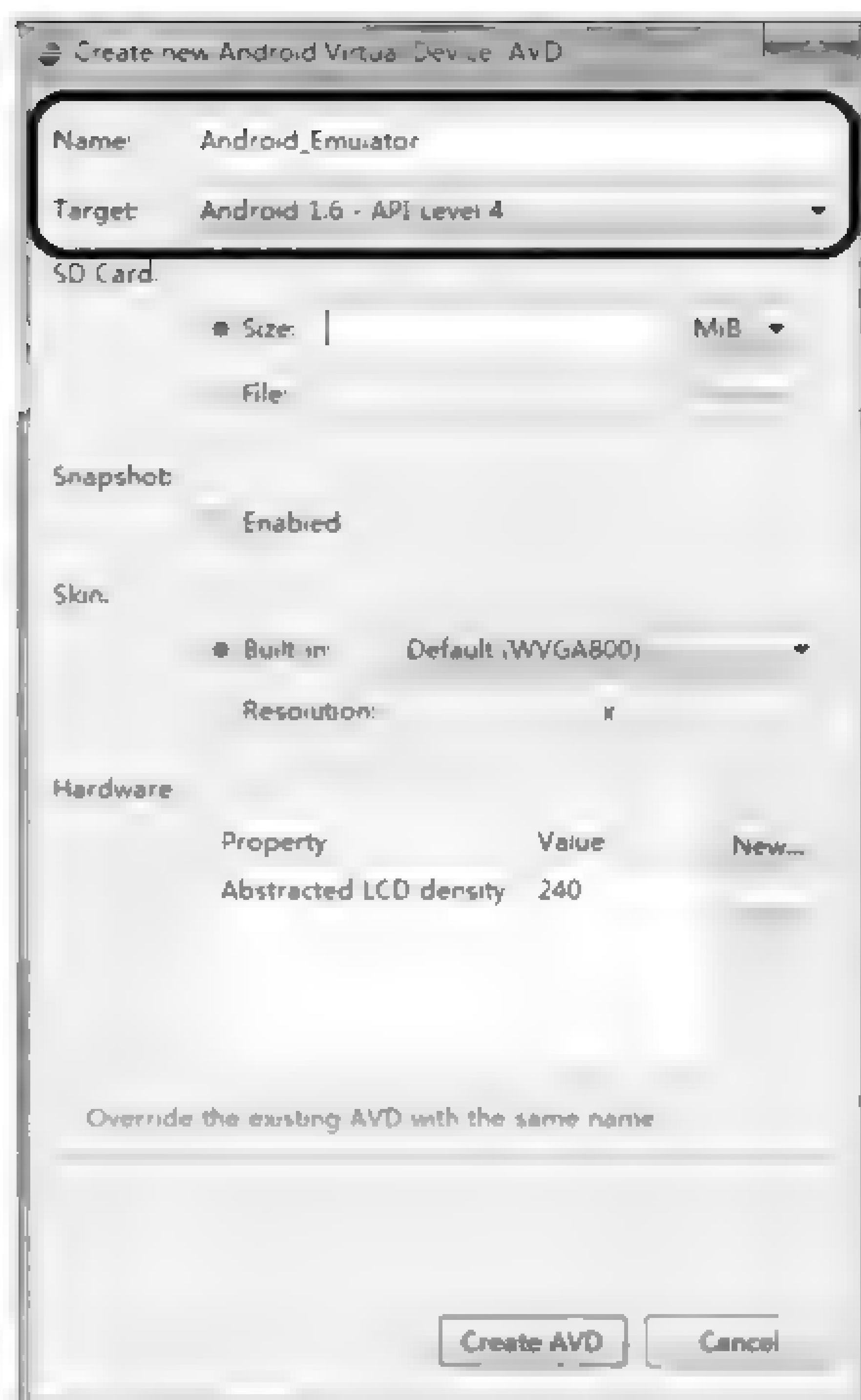


图 1-13 填写虚拟设备属性界面

1.2.3 SDK 版本更新

1. 更新 ADT 插件

首先更新 Eclipse 的 ADT 插件，具体步骤如下：

步骤1 单击 Eclipse 工具栏，打开“Help”菜单，然后选中“Check for Updates”选项，如图 1-14 所示。

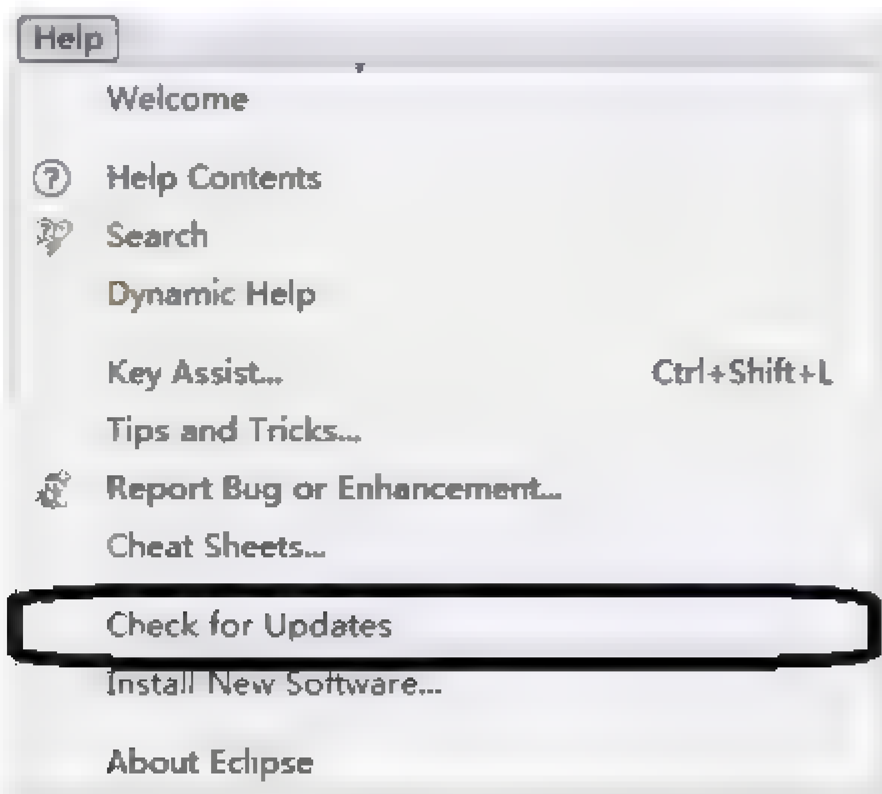


图 1-14 插件更新选项

步骤2 然后 Eclipse 会联网检查可更新的插件，如果有可更新的插件会进入“Available Updates”界面，如图 1-15 所示。

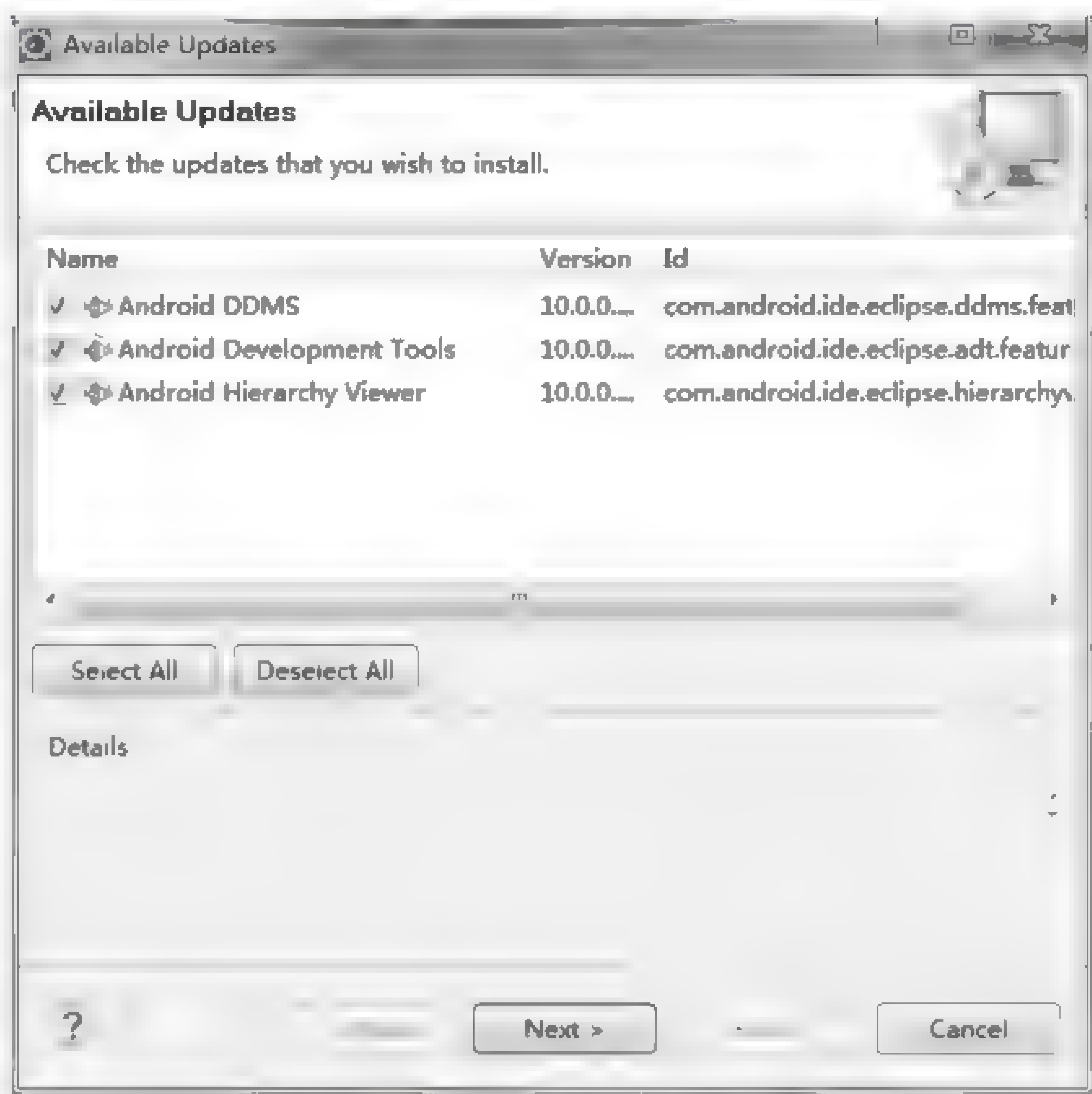


图 1-15 可更新插件选择界面

步骤3 在图 1-15 所示的界面中，可以看到当前最新 ADT 版本（version）是 10.0.0，选中需要更新的插件之后，单击“Next”按钮，最后单击“Finish”按钮即可。

2. 更新 Android SDK

当 ADT 更新完毕之后，就可以进行 Android SDK 的更新，具体步骤如下：

在 Eclipse 工具栏中选中“绿色机器人”图标进入 SDK&AVD 管理界面，单击左侧“Available packages”选项，然后选中右侧的“Android Repository”选项；选中后会自动联网检查可更新的 SDK 版本，之后选中需要升级的版本（如图 1-16 所示），接下来单击“Install Selectd”按钮，SDK 将自动下载安装。下载安装完毕后，重启 Eclipse 即可完成 SDK 的升级。

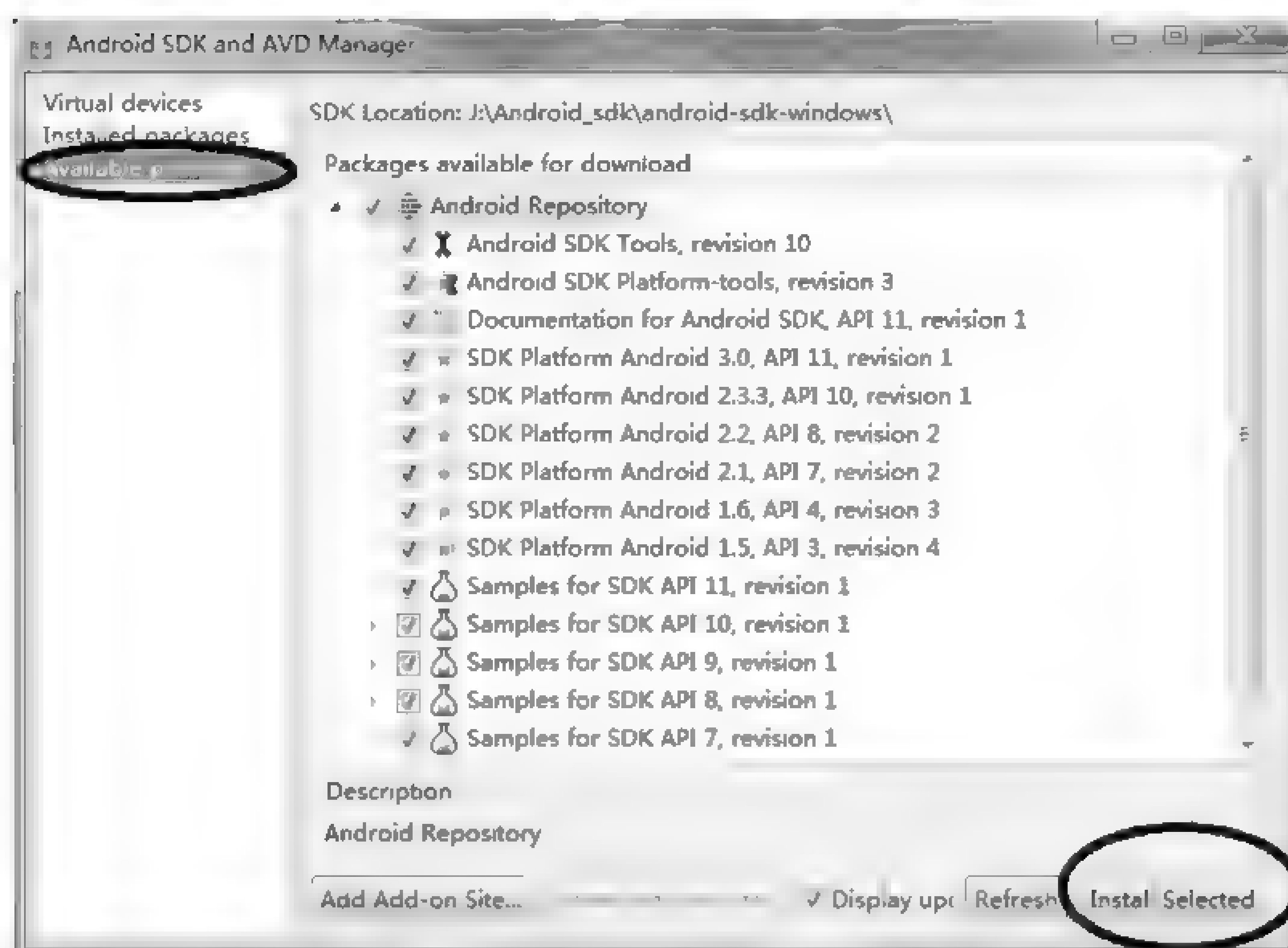


图 1-16 SDK 可更新版本选择界面

1.3 本章小节

本章简单介绍 Android 操作系统平台，并把重点放在 Android 开发环境的搭建上面。读者通过本章的学习，掌握 Android 开发环境的搭建和 SDK 版本更新的方法，为开始学习游戏应用开发打下工具基础。

第 2 章

Hello，Android!

从本章节可以学习到：

- ❖ 创建第一个 Android 项目
- ❖ 剖析 Android Project 结构
- ❖ AndroidManifest.xml 与应用程序功能组件
- ❖ 运行 Android 项目（启动 Android 模拟器）
- ❖ 详解第一个 Android 项目源码
- ❖ Activity 生命周期
- ❖ Android 开发常见问题



2.1 创建第一个 Android 项目

启动 Eclipse，依次单击“File→New→Project”菜单项，弹出以下界面（如图 2-1 所示），选中“Android”目录下的“Android Project”选项，单击“Next”按钮开始创建 Android 项目。

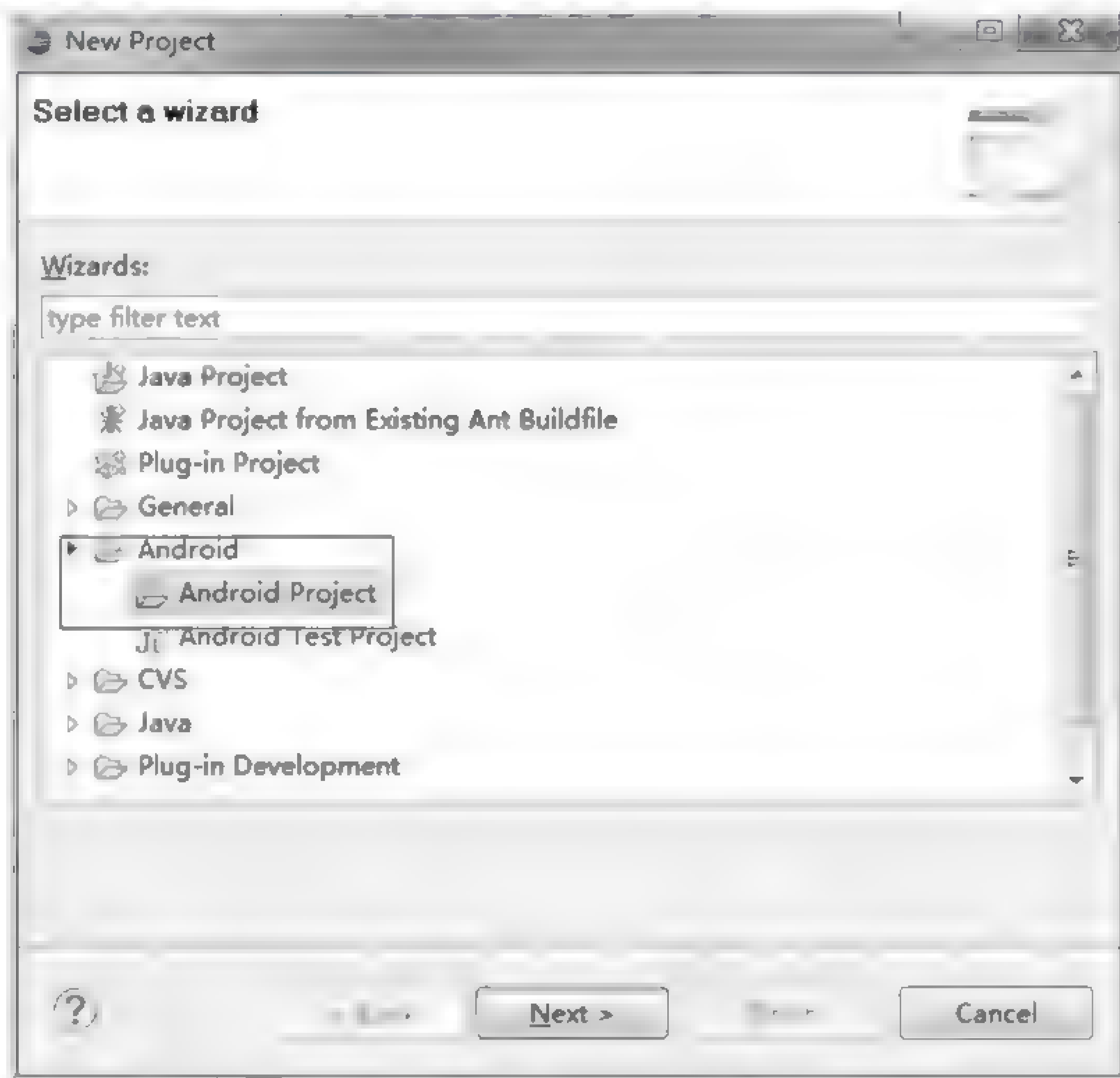


图 2-1 创建 Android 项目

然后弹出创建 Android 项目配置界面，如图 2-2 所示。在界面上的“Project Name”文本框填写“MyFirstProject”，在“Build Target”选项组中选中“Android 1.6”复选框，最后单击“Finish”按钮完成项目创建。

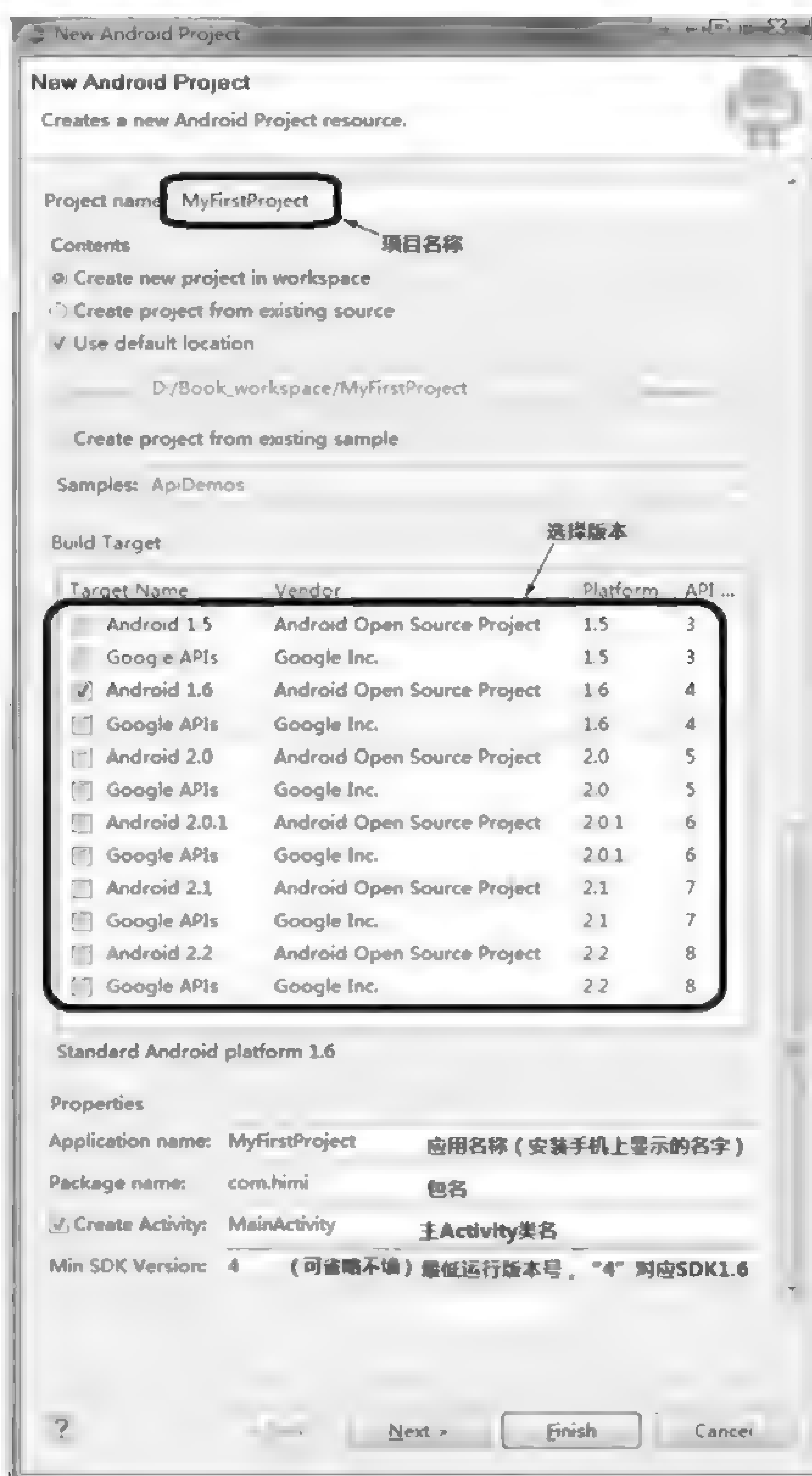


图 2-2　新建项目配置

2.2 剖析 Android Project 结构

在完成“MyFirstProject”项目创建之后，可以在“Pckage Explorer”视图中看到整个项目的结构，如图 2-3 所示。我们将在本节对 Android Project 的结构进行剖析。

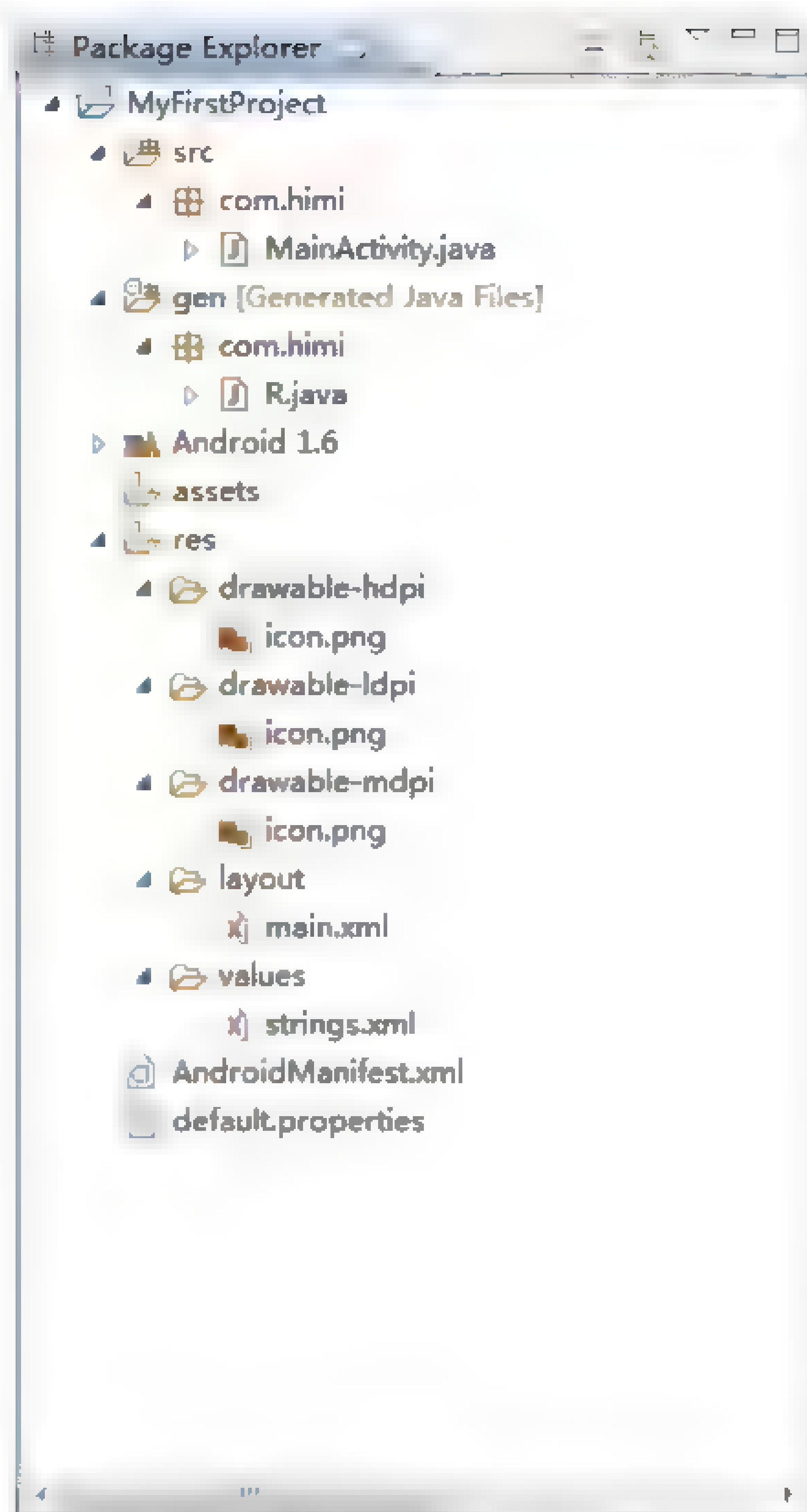


图 2-3　项目结构

（1）src 目录用来存放项目源代码（.java）。

（2）gen 目录下存放 R.java 文件，R.java 文件是该项目所有资源的索引文件，在建立项目时自动生成的，R.java 文件属于只读模式，不能更改，一般不会对其进行修改。R 类中包含很多静态类，其中这些静态类的名称都与 res 目录下的资源目录一一对应，如图 2-4 所示。

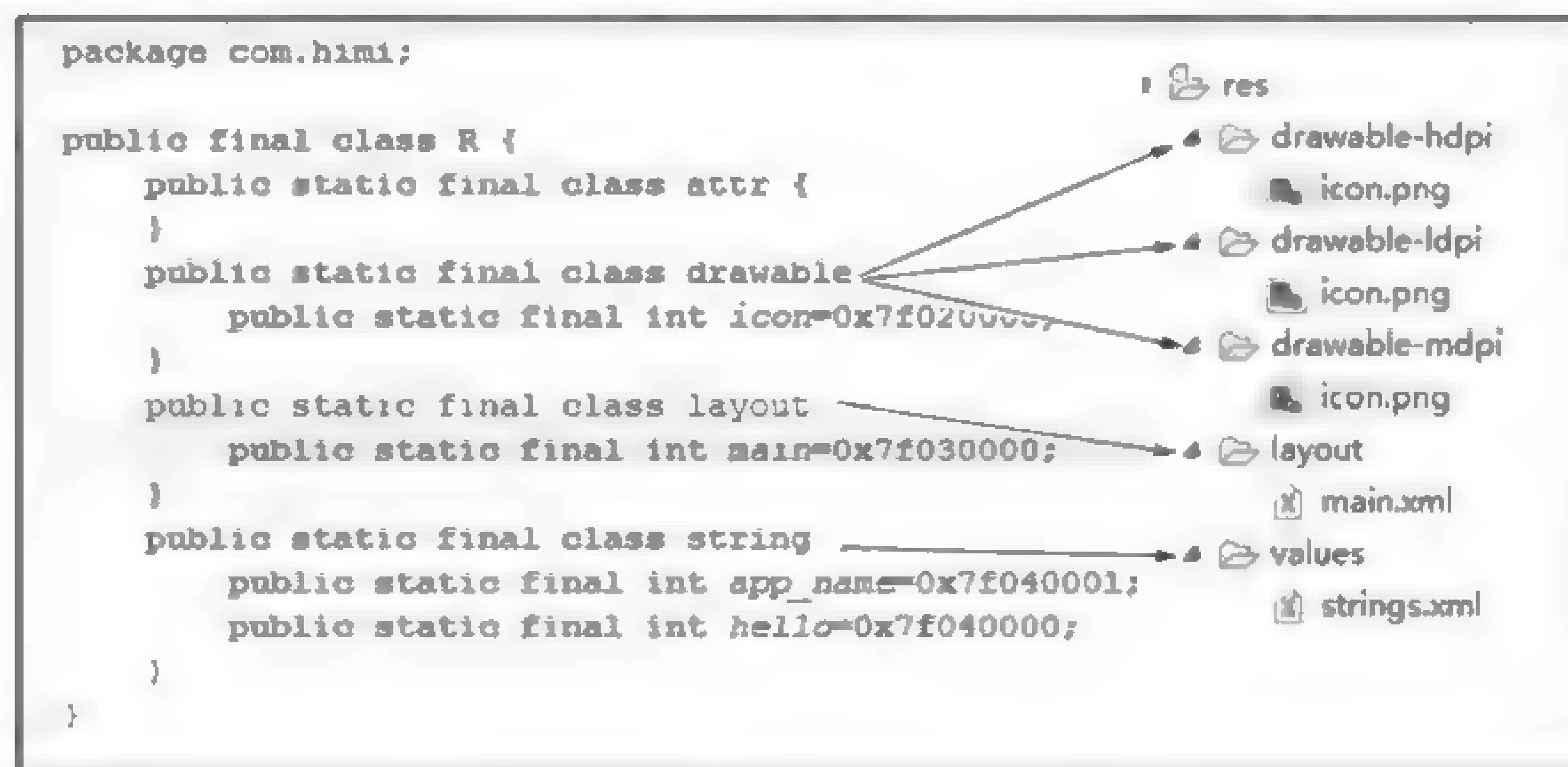


图 2-4　R 文件与资源文件的对应关系

假如在项目中新加一张图片放在 drawable 目录下，刷新项目，那么 R 资源文件就会更新，为其新添加的图片生成一个新的索引。

R 文件的作用和优点：

- 因为 R 文件会将所有资源生成索引，所以在项目中使用资源的时候会很便捷。
- 编译器在对项目进行编译的时候会检查 R 文件中的资源是否被使用，没有被使用的资源，编辑器不会编译到应用中，从而节省在手机中占用的内存。

（3）Android（Library）目录：此目录下的“android.jar”文件指向的是 Android SDK，是开发 Android 应用程序所用到的所有 API 函数库。

assets 目录是用来存放引用的外部资源，此目录与 res 目录最大的区别在于 res 路径下的资源文件都会在 R 资源文件中自动生成对应资源 ID，而 assets 目录下资源则不会。

（4）res 目录是用来存放项目中用到的资源文件，有 5 个默认子目录。

其子目录 drawable-hdpi、drawable-ldpi、drawable-mdpi 是用来存放图片资源的，如图片、图标（*.jpg、*.png 等）。

如果创建 Android 项目的时候选择的版本是 SDK 1.5 或者更低的版本，那么 res 目录下默认只有 3 个子目录，如图 2-5 所示。

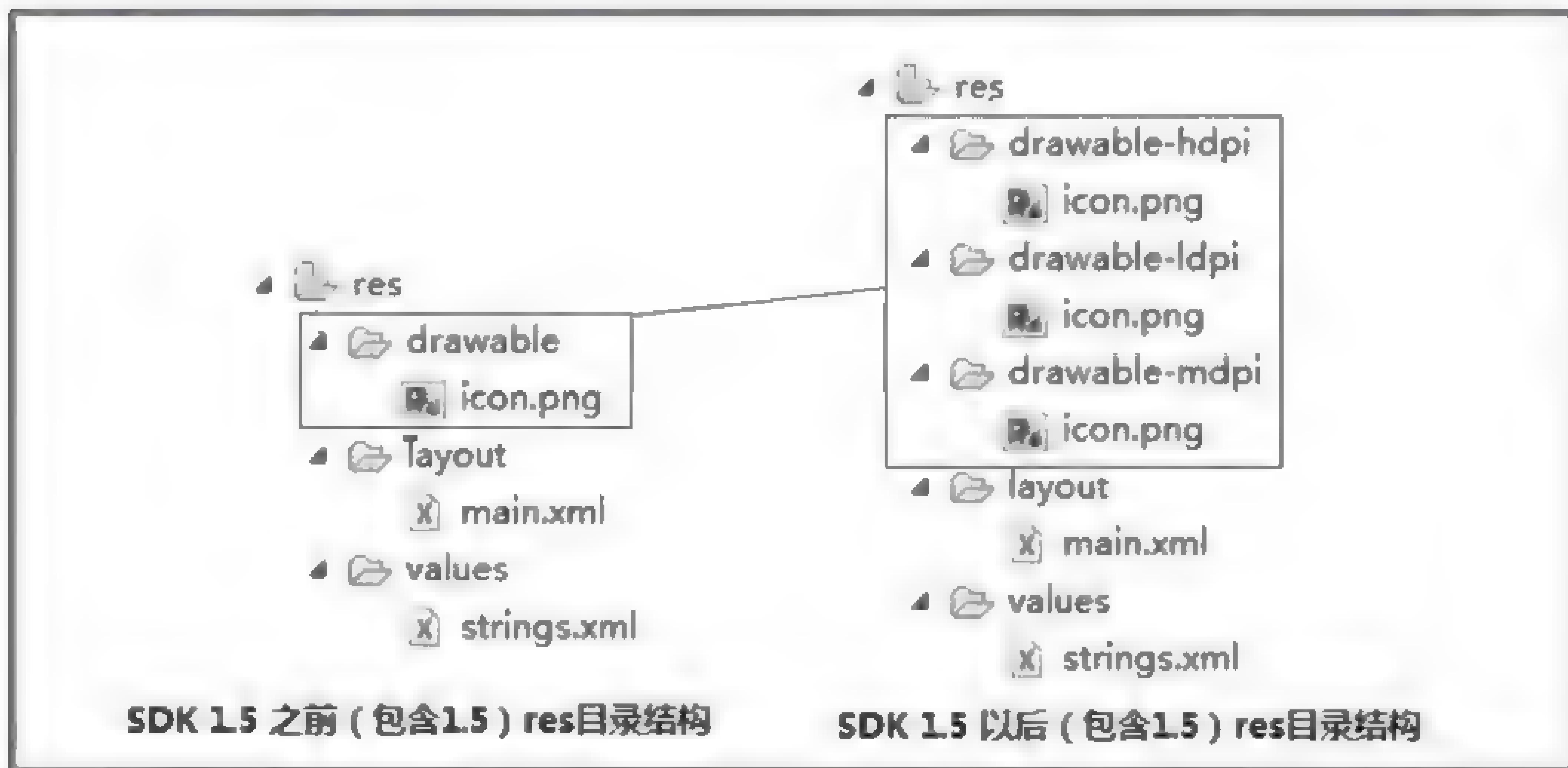


图 2-5　SDK 不同版本下 res 默认目录对比

大家还记得在创建虚拟设备的时候会设置一些模拟器的属性和参数，那么 Android 会根据创建的模拟器的分辨率参数分别选用对应目录下的图片资源，如果不小心放错了位置，那么也没关系，当 Android OS 在对应 drawable 目录下找不到资源时，会从其他两个子目录下进行寻找。

从 Android SDK 1.5 版本之后，res 目录中之前默认的 drawable 目录变成了三个类似的目录，这是 Google 为了方便开发者在开发应用时兼容不同机型屏幕所设计的一种让应用支持多

分辨率的形式，建议各文件夹根据需求均存放不同版本的图片，以下是其对应关系：

- drawable-hdpi：存放高分辨率的图片，例如 WVGA（480 × 800），FWVGA（480 × 854）；
- drawable-mdpi：存放中等分辨率的图片，例如 HVGA（320 × 480）；
- drawable-ldpi：存放低分辨率的图片，例如 QVGA（240 × 320）。

子目录 layout 用来存放布局文件以及 xml 格式的描述文件。

子目录 value 放置项目需要显示的各种文字，也可以存放多个*.xml 文件，存放不同类型的数据，比如 colors.xml（字体颜色）、styles.xml（显示类型）等等。

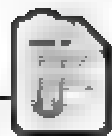
这里的 value 要延伸一下：假如应用需要发布到国外、中国台湾等地区，那么因为语言的差异，文字表达上肯定也要对其进行修改，如果应用做了本地化操作，就不用烦心这种事情。本地语言，简单的说，就是 Android 手机系统会根据手机系统语言默认使用程序中对应的目录资源文件的原理，如同上面的 drawable 目录一样，将不同的语言资源放置其目录下，例如把 value 文件夹扩展成以下形式，如图 2-6 所示。



图 2-6　本地化的 value 目录

当 Android 手机系统语言调整成英文的时候，会使用“value-en-rUS”目录，当是繁体语言的时候，会使用“value-zh-rTW”目录。这样一来，不同国家即使使用不同语言的手机系统，应用程序也照样适应。关于本地化后续文章会有更加详细的说明，这里就不多说了。

值得一提的是在 Android 系统中，图片、声音等资源名称不能使用大写英文字母！



注意

在讲解 res 目录的时候，一直在说明默认目录，其实在 res 中还有很多放置资源的目录，比如 raw（声音）等，也可以自定义子目录。

（5）AndroidManifest.xml 是当前项目的配置文件，其中包含编码格式、应用的 icon、程序的版本号以及指定该程序使用到的服务等等。这里需要注意，如果新添加一个 Activity，

就需要在这个文件中进行相应配置，然后才能调用此 Activity，设定 Activity 的属性也在此设定。

（6）default.properties 是记录项目工程的环境信息。

大家可能会对 Android 这种目录结构很疑惑，为什么要使用 xml 来进行开发，这里简单地说明一下。比如把项目中用到的系统组件都定义到 layout 布局文件中，而项目中所有用到的字符串都定义到 string.xml 中。一旦需要更新应用，那么无需再去 java 源码中修改和更新数据，而是直接对其 layout 和 string.xml 进行修改和更新就可以了；也就是说利用 xml 来进行开发可以更好的维护项目以及方便修改和更新程序。

2.3 AndroidManifest.xml 与应用程序功能组件

在每个 Android 应用程序中都必须具备 AndroidManifest.xml 这个文件，此文件定义了该程序的主要功能、处理的信息，以及执行的动作等等。这里对其进行详细说明，并通过 AndroidMainfest 对 Android 应用程序组件进行详细地介绍。

打开“MyFirstProject”项目下的 AndroidMainfest.xml 文件：

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.himi"
      android:versionCode="1"
      android:versionName="1.0">
   <application android:icon="@drawable/icon"
                android:label="@string/app_name">
      <activity android:name=".MainActivity"
                android:label="@string/app_name">
          <intent-filter>
             <action android:name="android.intent.action.MAIN" />
             <category
             android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

2.3.1 AndroidManifest 的 xml 语法层次

AndroidManifest 的 xml 语法层次如图 2-7 所示。



图 2-7　AndroidManifest 的 xml 语法层次

在上面的层次图中，在同一层的标签都是按照位置关系写在了一起，不同层的标签并不都是包含关系。

下面，根据当前“MyFirst Project”项目的 AndroidMainfest.xml 文件进行分析：

- 最外层的<mainfest>定义了软件的属性，如包路径、程序的版本、版次等等。
- 第 2 层<application>定义应用程序属性及功能；如 android:icon 指定了应用的 icon 图标，android:label 定义了应用程序的名称，并且声明功能<activity>活动。
- 第 3 层是定义功能组件的一层，如<activity>活动、<receiver>意图与广播接受、<service>服务、<provider>属性（内容）提供者，都必须定义在<application>中，所以这里定义了<activity>活动类以及活动类名称。
- 第 4 层<intent-filter>其功能如一个过滤器，后文将详细解释。
- 第 5 层中的<action>定义了意图动作的类型；<category>是意图的属性，这里的值 android.intent.category.LAUNCHER 表示启动应用程序的时候，意图将该 acctivity 显示在屏幕上；这种启动程序的意图活动只能有一个。

下面详细介绍 4 种应用程序的功能组件。

2.3.2　<activity>—Activity（活动）

Android 中一个 activity 就是一个用户界面，比如手机拨号界面、通讯录界面等都是活动。在应用程序中可以有一个或者多个活动，但是如果新建了一个活动，必须在 AndroidManifest.xml 中进行声明。

2.3.3　<receiver>—Intent（意图）与 Broacast Receiver（广播接收）

意图是描述动作的机制，在 Android 手机中几乎都有意图阶段。例如，当前有一个“发出一条信息”的意图（Intent），这时候如果应用程序需要发送信息，调用该意图即可。

既然有意图，当然也需要在应用程序中注册一个活动来处理该意图，那么接收意图可称为 Broadcast Receiver（广播接收）；举个例子，比如手机启动的时候会有一个系统发出“系

统启动”的意图，只要在应用程序中注册活动来接受此意图，就可以让其应用程序在手机启动的时候自动启动程序。因为在 Android 中有很多的意图，为了使接受具有针对性，可以在<intent-filter>中指定接收意图的动作<action>，因此< intent-filter >的作用就是一个意图的过滤器。

2.3.4 <service>—服务

简单来说，就是应用程序在后台运行的任务，无需显示界面、也不必跟用户进行交互，但是又需要长时间在后台进行运行；例如 Android 系统中的音乐播放器，当选定歌曲进行播放时，即使用户将播放器应用程序放入后台，然后打开其他程序，音乐也会照常播放，那么可以肯定播放器程序添加了 service 服务功能。

2.3.5 <provider>—Content Provider（内容提供者）

带有内容提供功能的应用程序所进行的动作是让使用者可以保存他们的信息或文件，例如：Android 手机联系人程序提供了一个内容提供者，任何要使用联系人信息的应用程序都可以共享内容提供者的所有信息，比如联系人的姓名、电话号、地址等等。

在 AndroidManifest.xml 文件中除了以上介绍的应用功能组件之外，Android 对某些关键操作和访问都是有权限来限制的。假如一个应用程序需要将一些数据保存到手机的 SDCard 中，那么就需要在 AndroidManifest 中设置添加写入 SD 卡权限，其语法标签是<uses-permisson>。例如在“MyFirstProject”项目中添加一条写入 SD 卡的权限代码，修改如下：

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.himi"
      android:versionCode="1"
      android:versionName="1.0">
   <application android:icon="@drawable/icon" android:label=
   "@string/app_name">
        <activity android:name=".MainActivity"
                  android:label="@string/app_name">
             …//代码省略
        </activity>
    </application>
<uses-permission android:name=
                         "android.permission.WRITE_EXTERNAL_STORAGE"/>
</manifest>
```

<uses-permission>与<application>在 AndroidManifest 语法中属于同一层次。

2.4 运行 Android 项目（启动 Android 模拟器）

运行 Android 项目的方式概括起来说有两种：

1. 直接运行 Android 项目

Eclipse 主界面上，在“Package Explorer”窗口中找到需要运行的项目，单击项目名称，然后单击鼠标右键选中“Run As->Run Configurations”选项，进入运行配置界面，如图 2-8 所示。



图 2-8　Run 配置　运行项目选择页面

在“Run Configurations”窗口的“Android”标签页中单击“Browse...”按钮，选中需要运行的项目，然后进入“Target”标签页中，如图 2-9 所示。

在“Run Configurations”窗口的“Target”标签页下，选择运行设备的运行方式：有手动和自动两种，然后勾选已创建的设备，最后单击“Run”按钮即可。

再次运行项目时，不必进入“Run Configurations”来重新设置，只需要在“Package Explorer”窗口中右键单击选中项目名，按照“Run As”→“Android Application”的步骤即

可执行应用。



图 2-9　Run 配置—运行设备设置页面

2. 先启动模拟器，再运行 Android 项目

在 SDK&AVD 管理界面（如第 1 章中的图 1-12 所示），选择已经建立好的 AVD，单击“Start”按钮即可启动模拟器（头次启动 Android 模拟器会有点慢）。

模拟器启动之后按照第一种运行项目的步骤操作即可。

运行的时候要注意一点：因为 Android 是向下兼容，所以运行项目的时候，前提是已经创建了高于或者等于当前项目的运行版本的模拟器，否则在“Target”设备一栏不会显示任何设备。

下面比较一下 SDK 1.5 模拟器与 SDK 1.6 模拟器的 UI，如图 2-10、图 2-11 所示，拿这两个来做对比是因为 1.6 以上版本（包含 SDK 1.6）的模拟器，外观 UI 基本相同。



图 2-10　Android SDK1.5（API-3）模拟器 UI

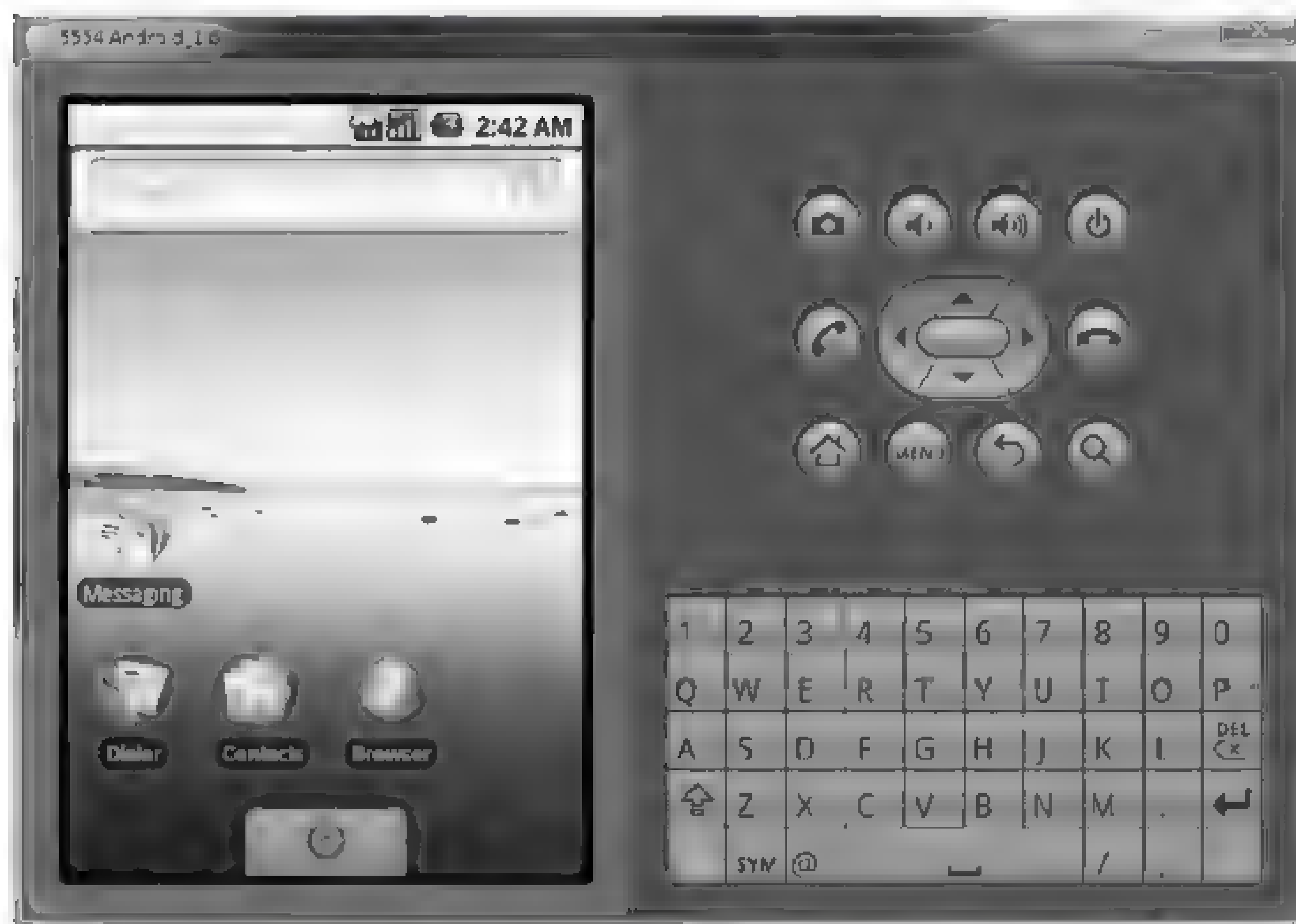


图 2-11　Android SDK1.6（API-4）模拟器 UI

2.5 详解第一个 Android 项目源码

运行“MyFirstProject”，在模拟器中显示的效果如图 2-12 所示。

程序的屏幕结构如图 2-13 所示。



图 2-12　MyFirstProject 运行效果图



图 2-13　运行程序的屏幕结构图

通过屏幕结构图可以看到，当程序运行起来之后，手机屏幕上不是只有“应用程序视图”，也会默认显示出“手机状态栏”和“应用名称”，至于如何全屏运行程序，后续文章会有详细说明，这里先来分析第一个 Android 程序的代码。

单击“MyFirstProject”项目下存放源码的 src 目录，打开 MainActivity.java 代码，如图 2-14 所示。

```
1  package com.himi;
2  import android.app.Activity;
3  import android.os.Bundle;
4  public class MainActivity extends Activity {
5      @Override
6      public void onCreate(Bundle savedInstanceState) {
7          super.onCreate(savedInstanceState);
8          setContentView(R.layout.main);
9      }
10 }
```

图 2-14　MainActivity 源码

第 1 行：	本类所在的包路径。
第 2~3 行：	引入相关类。
第 4 行：	创建一个类，并继承 Activity 类。
第 5 行：	“@Override”表示下面的 onCreate()函数（方法），是重写了基类 Activity 中的 onCreate()方法；如果没有这个标识，编译代码的时候会认为这是开发者自定义的函数。
第 6 行：	重写了 Activity 生命周期中的 onCreate()方法。
第 7 行：	调用父类的 onCreate()函数。
第 8 行：	利用当前 Activity 类的 setContentView()来显示布局。

通过上面逐行对 MainActivity 类中代码的解释，可以看出，主要起作用的代码就是第 8 行将 R.layout.main 传入 setContentView()方法中，通过 Activity 把布局显示在屏幕上。下面我们来看看如图 2-15 所示的 main.xml 布局文件中的代码：

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3      android:orientation="vertical"
4      android:layout_width="fill_parent"
5      android:layout_height="fill_parent"
6  >
7      <TextView
8      android:layout_width="fill_parent"
9      android:layout_height="wrap_content"
10     android:text="@string/hello"
11     />
12 </LinearLayout>
13
```

图 2-15　main.xml 源码

第 1 行：　描述 xml 的版本以及编码格式。

第 2 行：　定义布局形式，LinearLayout 为线性布局。

第 3 行：　设置布局位置放置的类型，即“vertical”垂直放置。

第 4 行：　设置布局的宽为填充类型，即填充屏幕。

第 5 行：　设置布局的高为填充屏幕。

第 6 行：　“>”布局基础属性设置结束，这里不是结束布局，到 12 行才是将布局结束。

第 7 行：　在布局中添加 TextView 组件。

第 8 行：　设置 TextView 组件的宽为填充类型。

第 9 行：　设置 TextView 组件的高为自适应类型，即高度根据其内容自动更改大小。

第 10 行：　设置 TextView 组件的文本内容。

第 11 行：　“/>”表示 TextView 设置结束，也可以写成“></TextView>”。

第 12 行：　整个线性布局设置结束。

对于布局、组件等属性的设置格式一般为：

```
<布局/组件名称
      android:属性=“属性类型”
              …
/>
```

第 10 行中的“@string/hello”，这里的“@string”表示索引 string.xml 中定义的字符串，“hello”则是在 string.xml 中定义字符串的变量名，对应关系如图 2-16 所示。

```
main.xml:
android:text="@string/hello"

string.xml
<string name="hello">Hello World, MainActivity!</string>
```

图 2-16　main.xml 中引用 string.xml 中的字符串示意图

到这里基本完成了第一个 Android 项目源码的分析，以及每个资源文件之间关联关系的介绍。可能大家现在对 MainActivity.java 这个类中的代码还是会一头雾水，下面就来详细介绍一下 Activity 的生命周期，相信通过 Activity 生命周期的讲述，会使大家思路清晰很多。

2.6 Activity 生命周期

在 Android 开发中，Activity 是非常重要的。Activity 主要负责创建和显示窗口，也可以把一个 Activity 理解成是一个显示的屏幕；在 Android 的应用中不是仅有一个 Activity，而是可以有多个 Activity 存在。因其重要性，开发 Android 务必熟悉 Activity 生命周期。

2.6.1　单个 Activity 的生命周期

下面我们先看一张官方给出的 Activity 生命周期图，如图 2-17 所示。

首先介绍 Activity 生命周期中的七个函数：

- onCreate：Activity 初次创建时被调用，一般在这里创建 view、初始化布局信息、将数据绑定到 list 以及设置监听器等等。如果 Activity 首次创建，本方法将会调用 onStart()；如果 Activity 是停止后重新显示，则将调用 onRestart()。
- onStart：当 Activity 对用户即将可见的时候被调用，其后调用 onResume()。
- onRestart：当 Activity 停止后重新显示的时候会被调用，然后调用 onStart()。
- onResume：当用户能在界面中进行操作的时候被调用。
- onPause：当系统要启动一个其他的 Activity 时调用（其他的 Activity 显示之前），这个方法被用来停止动画和其他占用 CPU 资源的事情。所以在这里应该提交保存那些持久数据，这些数据可以在 onResume()方法中读出。
- onStop：当另外一个 Activity 恢复并遮盖住当前 Activity，导致其对用户不再可见时调用。一个新 Activity 启动、其它 Activity 被切换至前景、当前 Activity 被销毁时都会调用此函数。如果当 Activity 重新回到前景与用户交互时会调用 onRestart()，如果

Activity 将退出则调用 onDestory()。

- onDestory：在当前的 Activity 被销毁前所调用的最后一个方法，当进程终止时调用（对 Activity 直接调用 Finish 方法或者系统为了节省空间而临时销毁此 Activity 的实例）。

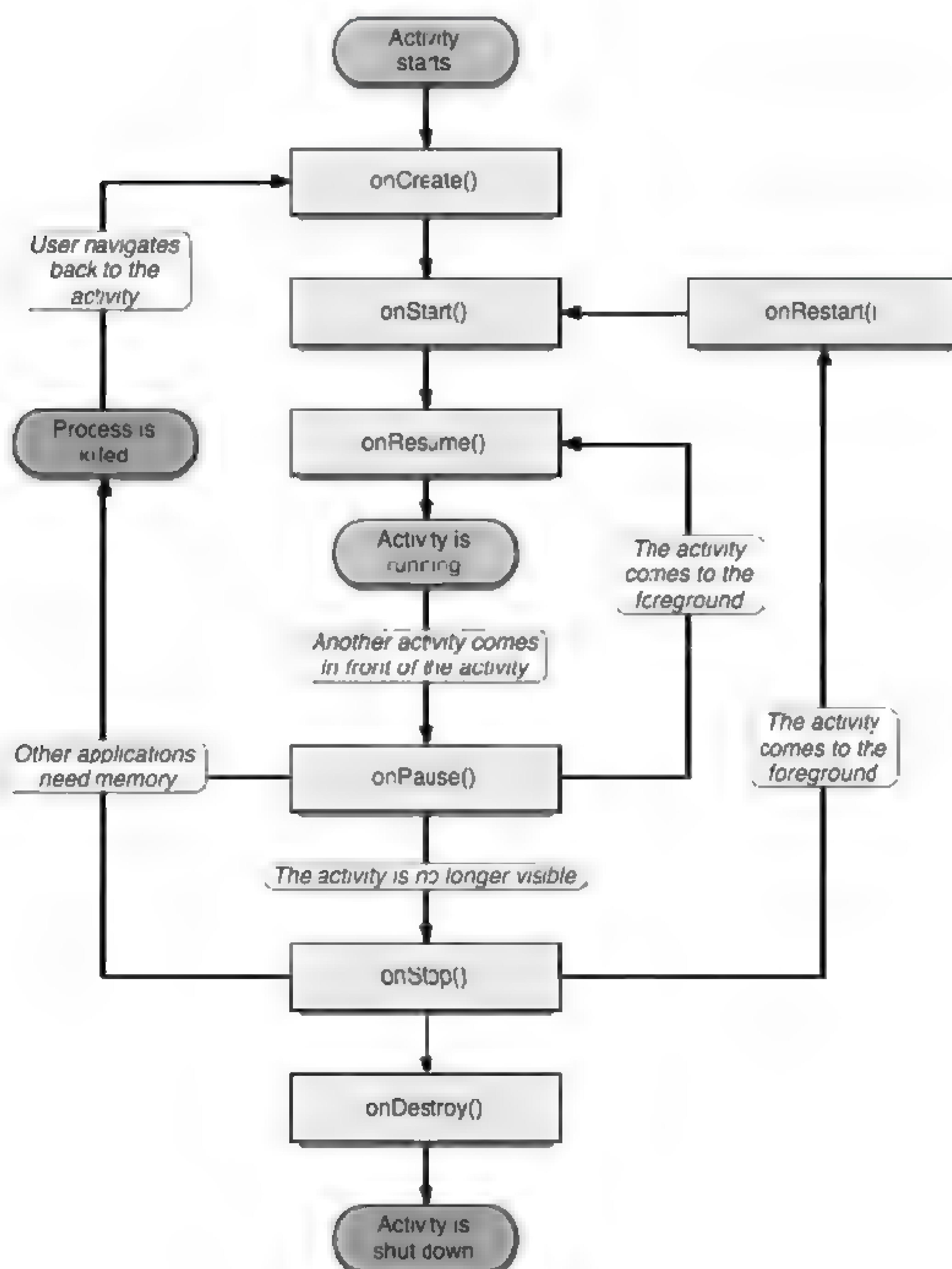


图 2-17　Activity 生命周期图

为了让大家更清楚地看到 Activity 生命周期的变化，下面我们把“MyFirstProject”项目修改一下，对应的源代码为“2-1（Activity 生命周期）”，运行效果如图 2-18 所示（这里为了给大家讲解 Activity 生命周期，所以如何修改的代码暂且不做说明）。



图 2-18 MyFirstProject 修改后的运行效果图

首先在“MyFirstProject”项目中新添加了一个类“OtherActivity.java”，这个类也是一个Activity；然后两个 Activity 类中都重写除了默认生成的“OnCreate”函数之外的六个生命周期函数，并在每个生命周期函数中添加一句 Log 打印语句，然后用“MainActivity”的 Activity 打开新添加的“OtherActivity”类中的 Activity，从而观察生命周期的变化和七个函数之间的调用顺序。

Activity 类中的七个生命周期函数以及每个函数中打印的内容如下：

```
@Override
public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.main);
}
@Override
protected void onDestroy() {
      super.onDestroy();
      Log.v("MainActivity", "onDestroy");
}

@Override
protected void onPause() {
      super.onPause();
      Log.v("MainActivity", "onPause");
}

@Override
protected void onRestart() {
      super.onRestart();
      Log.v("MainActivity", "onRestart");
}

@Override
protected void onResume() {
      super.onResume();
      Log.v("MainActivity", "onResume");
}

@Override
```

```
protected void onStart() {
        super.onStart();
        Log.v("MainActivity", "onStart");
}

@Override
protected void onStop() {
        super.onStop();
        Log.v("MainActivity", "onStop");
}
```

运行项目，在 LogCat 视图中观察打印信息：

（1）首先启动项目进入 MainActivity 类，打印信息如图 2-19 所示。

```
03-06 09:29...  V  254  MainActivity        onCreate
03-06 09:29...  V  254  MainActivity        onStart
03-06 09:29...  V  254  MainActivity        onResume
```

图 2-19　启动项目的打印信息

（2）当单击手机上的“Back”按键，打印信息如图 2-20 所示。

```
03-06 09:33...  V  254  MainActivity        onPause
03-06 09:33...  V  254  MainActivity        onStop
03-06 09:33...  V  254  MainActivity        onDestroy
```

图 2-20　单击“Back”按键的打印信息

（3）“Back”后重新单击程序图标进入，打印信息如图 2-21 所示。

```
03-06 09:37...  V  254  MainActivity        onCreate
03-06 09:37...  V  254  MainActivity        onStart
03-06 09:37...  V  254  MainActivity        onResume
```

图 2-21　重新单击程序图标的打印信息

（4）当单击手机上的“Home”（小房子）按键，打印信息如图 2-22 所示。

```
03-06 09:40...  V  254  MainActivity        onPause
03-06 09:40...  V  254  MainActivity        onStop
```

图 2-22　单击“Home”按键的打印信息

（5）“Home”后重新单击程序图标进入，打印信息如图 2-23 所示。

```
03-06 09:44...  V  254  MainActivity        onRestart
03-06 09:44...  V  254  MainActivity        onStart
03-06 09:44...  V  254  MainActivity        onResume
```

图 2-23　重新单击程序图标的打印信息

以上 5 种状态，详细地诠释了一个 Activity 的生命周期。这 5 种状态中值得注意的是切入后台“Back”和“Home”的区别，如图 2-24 所示。

操作	“Back”方式	“Home”方式
将Activity 切入后台：	onPause onStop onDestroy	onPause onStop
将Activity 从后台重新回到前台	onCreate onStart onResume	onRestart onStart onResume

图 2-24　Back 与 Home 的异同之处

2.6.2　多个 Activity 的生命周期

一个 Android 应用程序，是可以存在多个 Activity 的；下面就来看当存在两个（或者多个）Activity 的时候，每个 Activity 的生命周期状态变化。

修改后的项目运行效果如图 2-18 所示，添加了一个名为 button 的按钮，单击这个按钮就会打开另外一个 Activity（OtherActivity）。

（1）在 MainActivity 中，单击“button”按钮，在 LogCat 视图中显示如图 2-25 所示的信息。

```
03-06 ...  V  483  MainActivity    onPause
03-06 ...  V  483  OtherActivity   onStart
03-06 ..   V  483  OtherActivity   onResume
03-06 ..   V  483  MainActivity    onStop
```

图 2-25　单击“button”按键的打印信息

（2）当 OtherActivity 打开之后，单击手机上的“Back”按钮，在 LogCat 视图中显示如图 2-26 所示的信息。

```
03-06  .   V  508  OtherActivity   onPause
03-06  ..  V  508  MainActivity    onRestart
03-06  ..  V  508  MainActivity    onStart
03-06  ..  V  508  MainActivity    onResume
03-06 ...  V  508  OtherActivity   onStop
03-06  .   V  508  OtherActivity   onDestroy
```

图 2-26　单击“Back”按键的打印信息

这里，可以设置 OtherActivity 的两种不同展示类型，如图 2-27 所示。

每个 Activity 都可以设置它的主题风格（模式），图 2-27 所示是 OtherActivity 的两种主

题风格截图，左边是默认的主题风格，右边把风格设置成了 Dialog（对话框）类型。



图 2-27　OtherActivity 的两种不同展示类型

为什么要提到这个问题，因为在多个 Activity 之间进行跳转还需要注意的就是：当打开另外的一个 Activity，我们要清楚另外一个 Activity 是否能完全遮挡住当前的 Activity 视图，具体区别如图 2-28 所示。

Activity (A) 打开Activity (B)

	Yes　(默认主题模式)		No　(Dialog主题模式)	
B的视图是否能完全覆盖A	MainActivity OtherActivity OtherActivity MainActivity	onPause onStart onResume onStop	MainActivity OtherActivity OtherActivity	onPause onStart onResume

图 2-28　Activity 主题模式不同对比图

从图 2-28 可以明显地看出，当打开另外一个 Activity 的时候，还要分析这个新打开的 Activity 会不会将当前 Activity 完全覆盖，如果不能完全覆盖，则在打开这个新的 Activity 之后，之前的 Activity 就不会调用 on Stop()这个生命周期函数了。

以上两种状态是从一个 Activity 跳转到另外一个 Activity 的生命周期流程，但是这里要注意：当第一个 Activity 跳转到另外一个 Activity 的时候，并没有在代码中手动关闭第一个 Activity。再来看看当打开另外一个 Activity 的时候，手动关闭之前的 Activity 时，两个 Activity 的生命流程（在一个 Activity 中，只要调用 finish()函数，即可退出当前 Activity）。

（1）在 MainActivity 中单击“button”按钮，打印信息如图 2-29 所示。

区别很明显，在打开 OtherActivity 时，如果程序中手动关闭了第一个 MainActivity，那么在打开另外一个 Activity 之后，MainActivity 在调用 onStop 之后会多调用一个 onDestory 的状态。

```
03-06 ...   V   534   MainActivity        onPause
03-06 ...   V   534   OtherActivity       onStart
03-06 ...   V   534   OtherActivity       onResume
03-06 ...   V   534   MainActivity        onStop
03-06 ...   V   534   MainActivity        onDestroy
```

图 2-29　单击“button”按钮的打印信息

（2）当 OtherActivity 打开之后，单击手机上的“Back”按钮，打印信息如图 2-30 所示。

```
03-06 ...   V   534   OtherActivity       onPause
03-06 ...   V   534   OtherActivity       onStop
03-06 ...   V   534   OtherActivity       onDestroy
```

图 2-30　单击“Back”按钮的打印信息

这个不用多做解释，因为 MainActivity 在 OtherActivity 之后就关闭掉了，所以程序中只有 OtherActivity 存在，这时候单击“Back”的生命周期流程就如同单个 Activity 的“Back”执行的生命周期流程。

2.6.3　Android OS 管理 Activity 的方式

前面介绍了 Activity 的七个生命周期函数，还详细分析了在多个 Activity 之间跳转时，每个 Activity 的生命周期流程。既然每个或者多个 Activity 都可以看成是一个应用程序，那么 Android OS 是如何管理这些 Activity 的呢？

对于 Acticity 的管理，Android 底层是用堆栈来存放 Activity 的，也就是说后打开的 Activity 将放入栈顶，显示在屏幕的最上层，而之前 Activity 则会被新打开的 Activity 覆盖。例如一个程序正在运行，突然手机来电，Android 接受到来电广播后会打开一个接听电话的 Activity 放入堆栈的栈顶，这样一来运行的程序就会被接听电话的 Activity 所覆盖。

2.7　Android 开发常见问题

2.7.1　Android SDK 与 Google APIs 创建 Emulator 的区别

在利用 AVD 进行创建 Android Emulator 时，要求选择 SDK 的版本，如图 2-31 所示，可以看到每个 SDK 版本都对应一个 Google APIs。

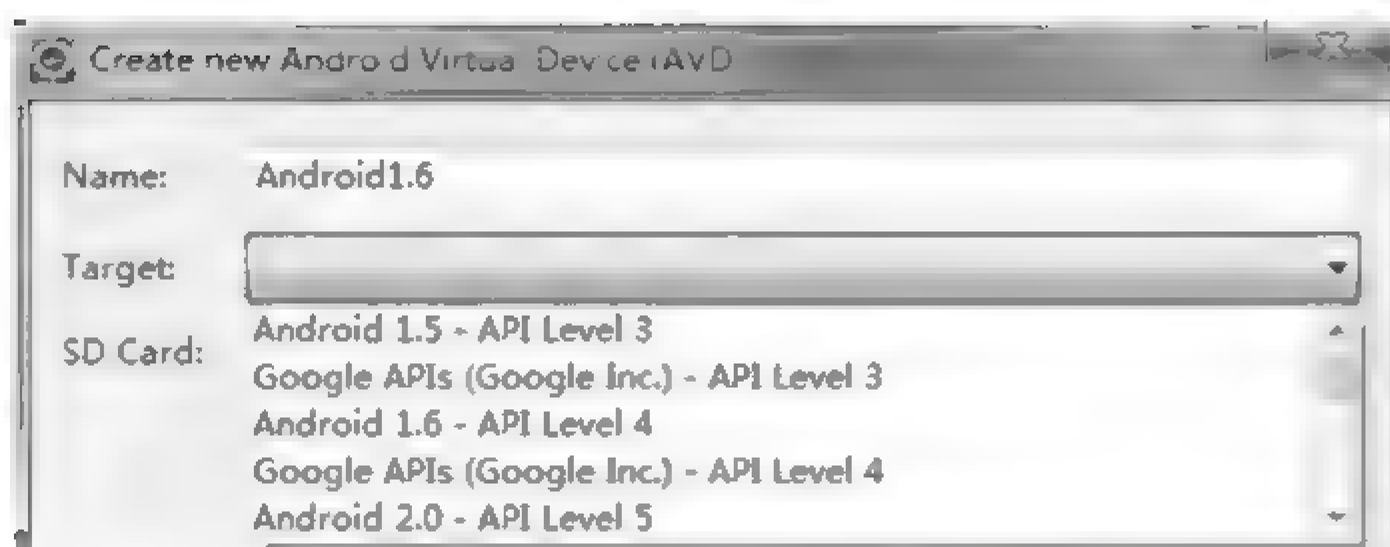


图 2-31 Android SDK 与 Google APIs 对应

如图 2-31 所示，Android 版本都对应有一个 Google APIs。其实两种形式创建的模拟器是没有差别的，只是在 Google APIs 中，Google 把自己提供的程序，如 Google Map 放在了 Google APIs 创建出来的模拟器中。因此，如果要开发 Google Map 等一些 Google 专属的应用程序，就必须选用 Google APIs 创建出的模拟器。

2.7.2 将 Android 项目导入 Eclipse

有时需要从外部导入一个 Android Project，有两种导入方式：新建 Android 项目时导入和直接导入。本书中用到的光盘中的所有项目例子，都可以通过这两种方式导入。

1. 新建 Android 项目时导入方式

按照 2.1 节讲述的创建 Android 新项目的顺序，到达图 2-2 所示的“新建项目配置”这一步，不填写任何信息，按照图 2-32 所示的操作。

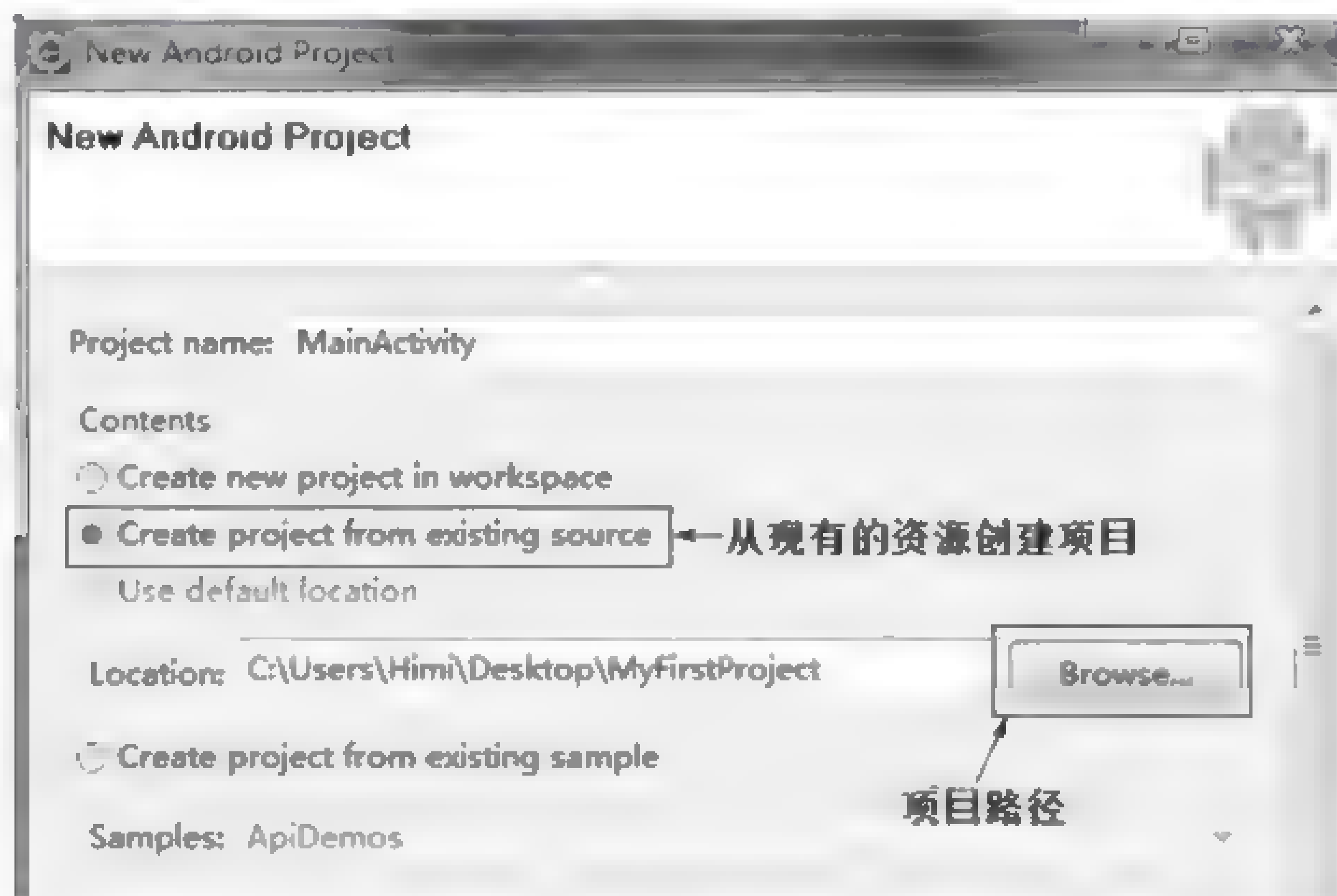


图 2-32 新建项目时导入项目

先选中“Create project from existing source”单选按钮，然后单击“Browse...”按钮选择项目路径，最后单击最下方的“Finish”按钮即可。

2. 直接导入方式

单击 Eclipse 菜单栏上的“File→Import”进入导入方式选择界面，如图 2-33 所示。

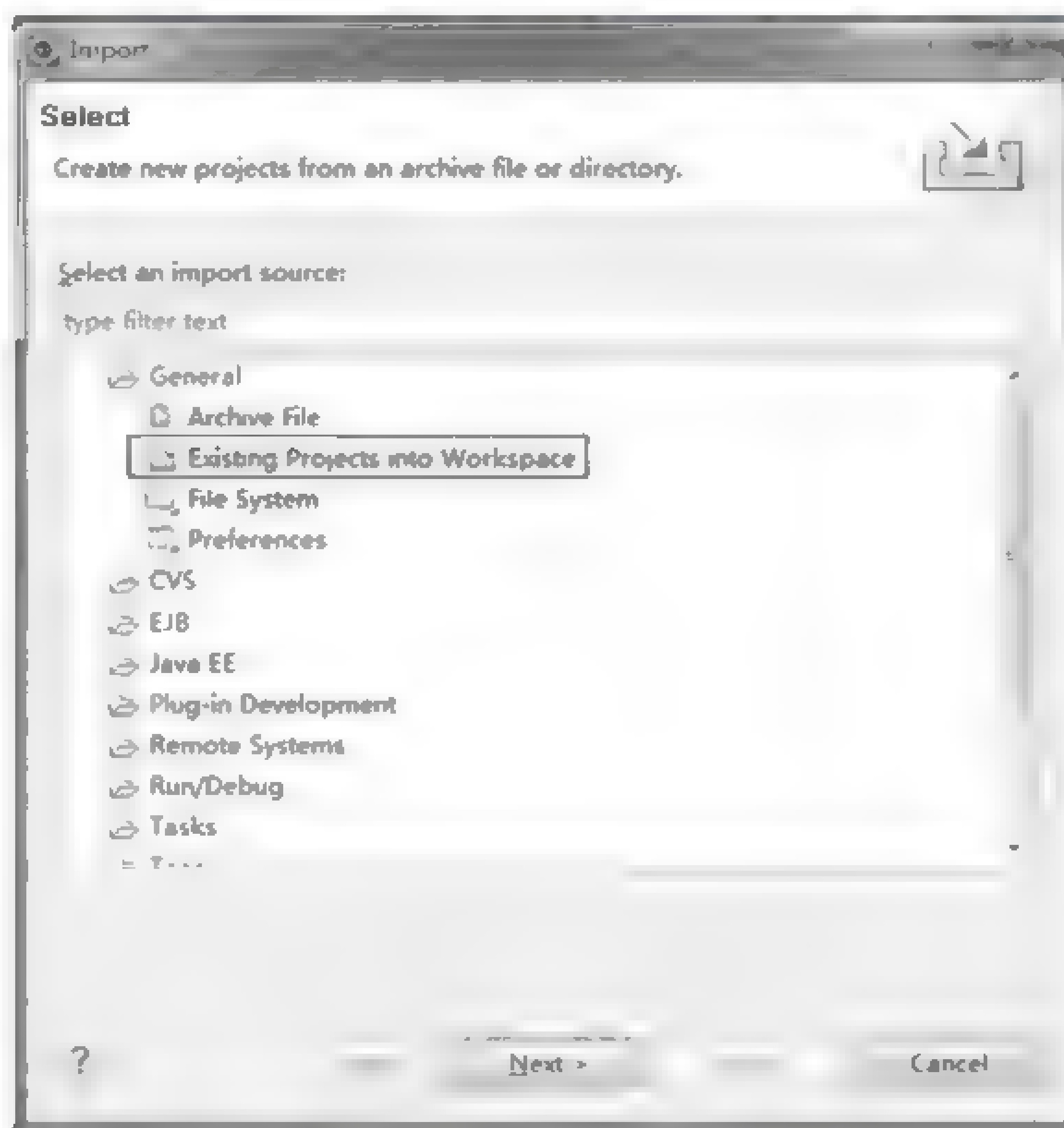


图 2-33　导入方式选择界面

选中“Existing Project into Workspace”选项，然后单击“Next”按钮进入导入项目界面，如图 2-34 所示。

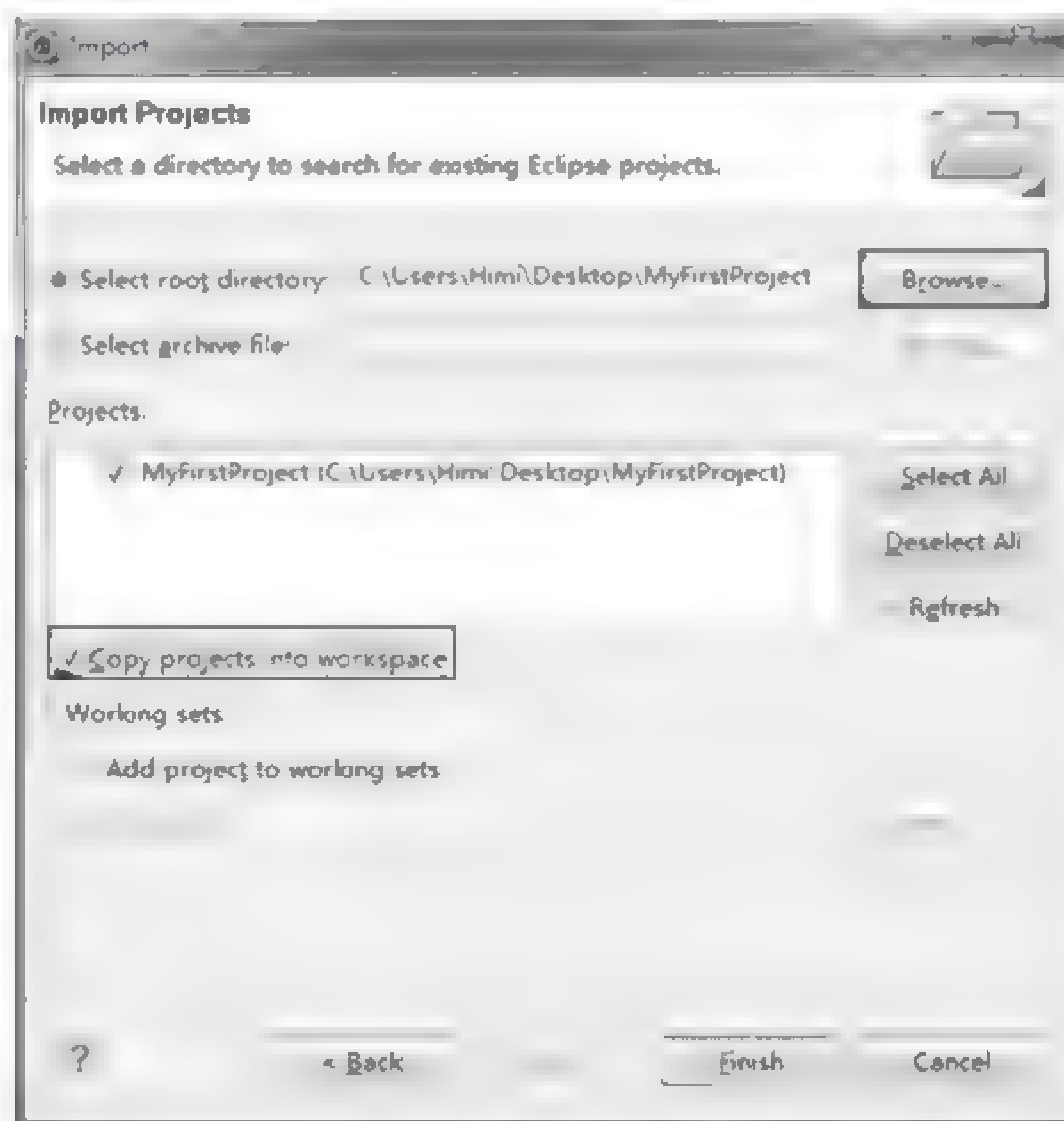


图 2-34　项目导入界面

单击“Browse…”按钮选择项目路径即可，当前界面的“Copy projects into workspace”复选框是否选中是可选操作，选中表示将导入的项目拷贝到 Eclipse 工作目录中，不选中则默认修改的是导入的项目。

2.7.3 在 Eclipse 中显示 Android 开发环境下常用的 View 窗口

在 Eclipse 菜单一栏中单击“Windows→Show View→Other”打开“Show View”窗口，如图 2-35 所示。

在图 2-35 所示的“Show View”窗口中，单击“Android”展开列表项，常用到的视图有 3 个：“Devices”、“File Explorer”和“LogCat”。

“Devices”窗口如图 2-36 所示。其中，图标 1 用于选中程序时，单击此图标即可进入 Debug 模式进行调试；图标 2 用于选中程序时，单击此图标即消亡程序；图标 3 用于截图，截取当前模拟器/真机屏幕。



图 2-35 选择视图 View



图 2-36 Devices 视图

“File Explorer”视图如图 2-37 所示。如果当前设备没有安装 SD 卡，则不会有“sdcard”一项。图中，图标 1 用于当选中文件时单击此图标表示导出文件；图标 2 用于当选中文件时单击此图标表示导入文件；图标 3 用于当选中文件时单击此图标表示删除此文件。

“LogCat”窗口如图 2-38 所示，这个窗口用于显示项目运行时的打印信息等。

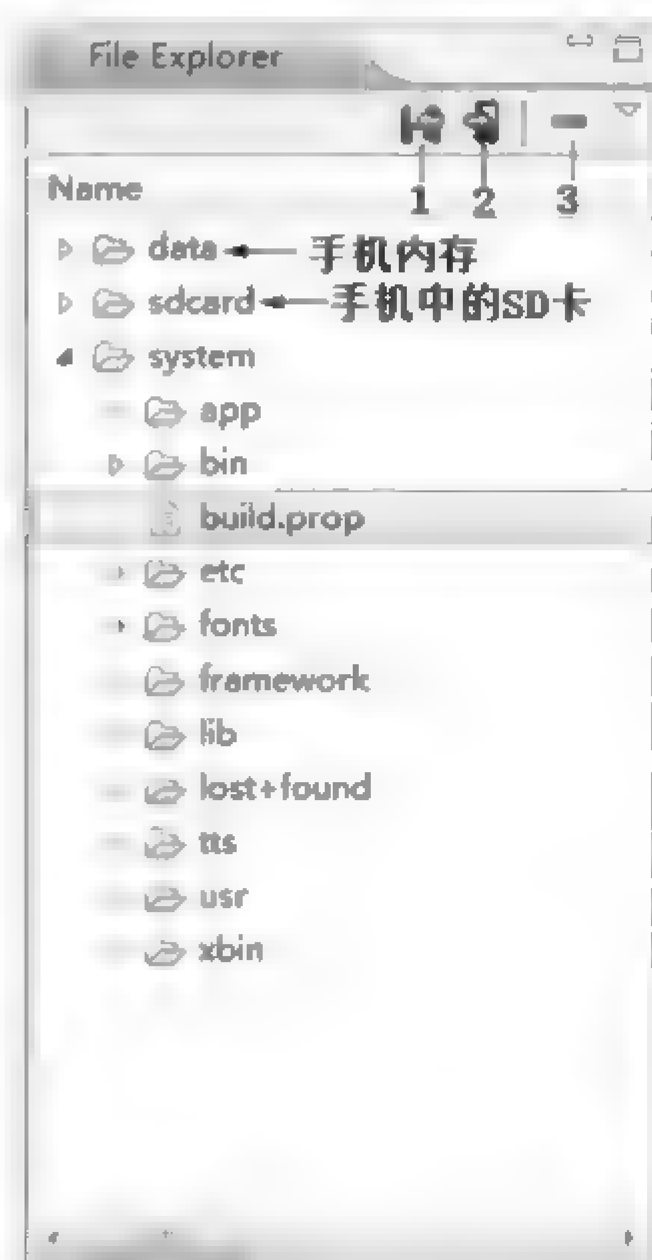


图 2-37　File Explorer 视图

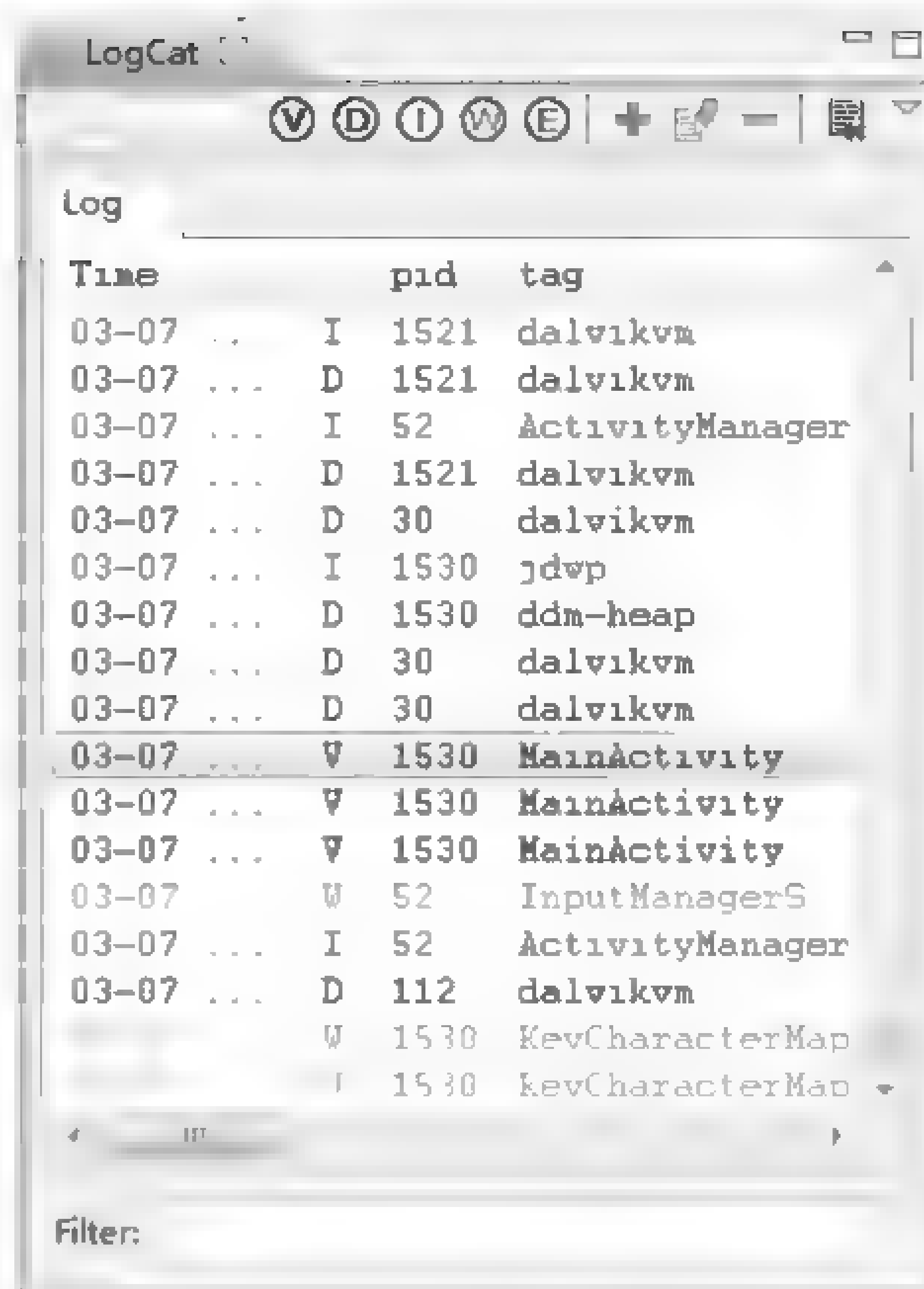


图 2-38　LogCat 视图

2.7.4　在 Eclipse 中利用打印语句（Log）调试 Android 程序

在调试程序时，偶尔会添加打印信息来掌握当前程序的运行状态。之前在介绍 Activity 生命周期的时候，已经使用了打印语句 Log.v()，这里简单介绍一下 Log 打印语句。

在 Android 中一般常用的打印语句有五种：

- Log.v（黑色）：任何消息都会输出，一般都利用这个打印来进行观察程序运行状况；
- Log.i（绿色）：输出的是提示性的信息；
- Log.d（蓝色）：只输出 Debug 的信息；
- Log.w（黄色）：输出警告信息；
- Log.e（红色）：输出错误信息。

一般情况下，使用 Log.v()进行打印。要注意一点，通常执行打印语句会在“LogCat”视图中看到打印出的信息，但有时候会发现程序中明明写了输出语句，而且也确实能执行到，可是“LogCat”视图中就是不显示打印的信息，这种情况的解决办法就是：在“Devices”视图中单击运行的程序，“LogCat”视图中就会显示打印的信息了。这是因为“LogCat”默认显示输出的是整个模拟器的运行状态，只有在“Devices”视图中单击（指定）到程序上，“LogCat”视图才会输出程序的打印信息。

2.7.5 在 Eclipse 中真机运行 Android 项目

将 Android 手机通过 USB 线连接到电脑上，然后在手机上选中“USB 调试”选项，之后在电脑上安装对应手机的驱动，这样在 Eclipse 的“Devices”视图中就会显示出真机设备了。如果想运行项目在手机上，只要在“Run Configurations”窗口（如图 2-9 所示）下的“Target”页面下选中“Manual”（手动选择运行设备），然后在 Run 的时候，手机设备如果正常连接，将会弹出“Android Device Chooser”窗口，如图 2-39 所示。

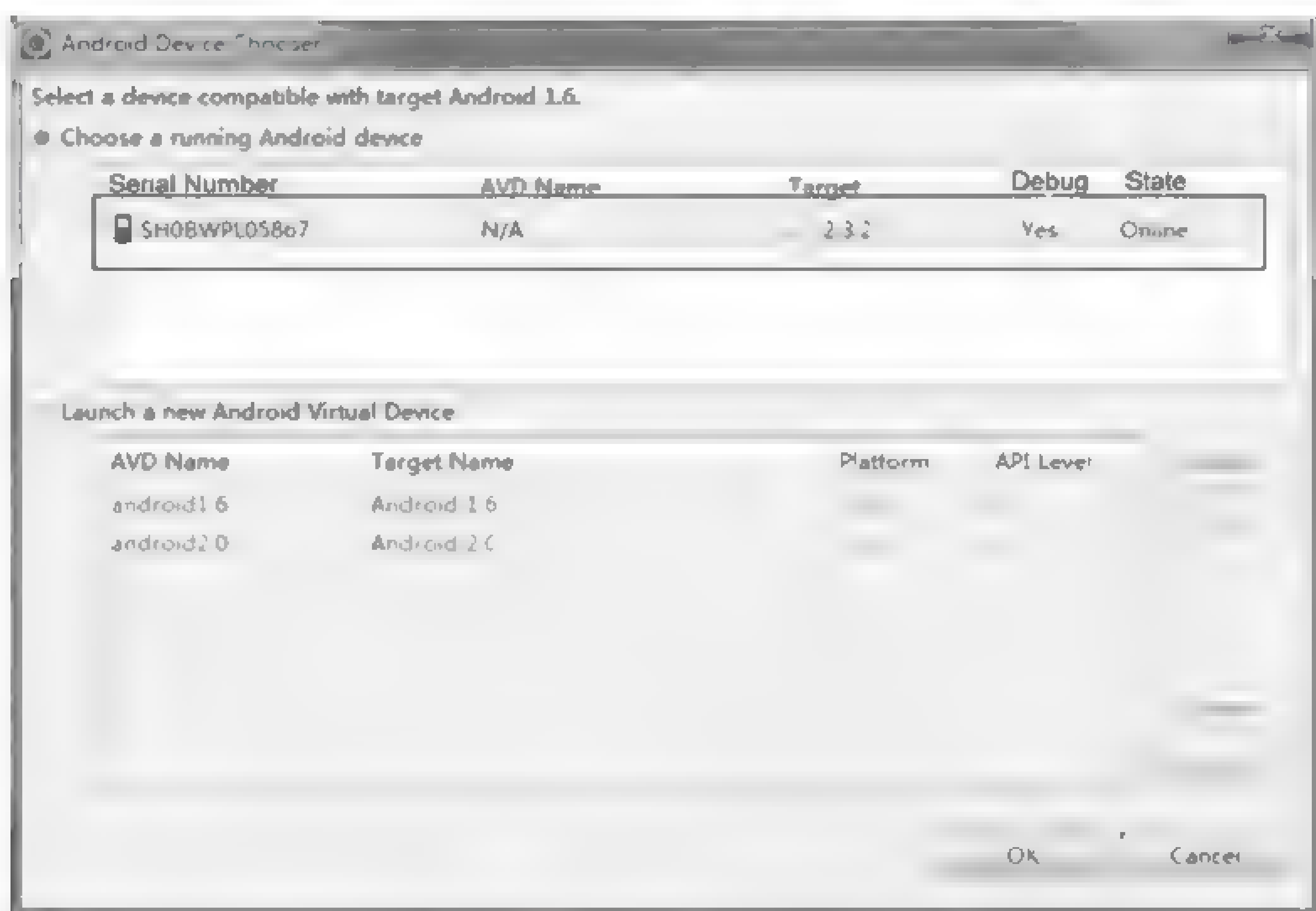


图 2-39 真机运行 Android 项目

在“Android Device Chooser”窗口中，选中真机设备，最后单击“OK”按钮运行。

2.7.6 设置 Android Emulator 模拟器系统语言为中文

默认模拟器系统语言为英文，设置系统语言的具体步骤为：单击运行设备上的“Menu→Settings→Locale & text→Select locale”菜单项，然后选择“简体中文”即可设置系统语言为中文。

2.7.7 切换模拟器的输入法

切换模拟器的输入法的操作步骤如图 2-40 所示。单击输入控件直到弹出“Edit text”对话框后释放鼠标，然后单击“Input Method”菜单进入选择输入法窗口界面，选中“谷歌拼音输入法”即可输入中文。



图 2-40 切换输入法流程图

2.7.8 模拟器中创建 SD Card

游戏开发中有时会保存一些数据到 SD Card 中，或是从 SD Card 中获取数据。本小节将介绍在 Android Emulator（模拟器）中如何创建 SD Card。

在模拟器上创建 SD Card 其实很简单，只需要在之前利用 AVD 创建 Android 模拟器的时候，稍微调整配置就可以了，具体方法为（如图 2-41 所示）：

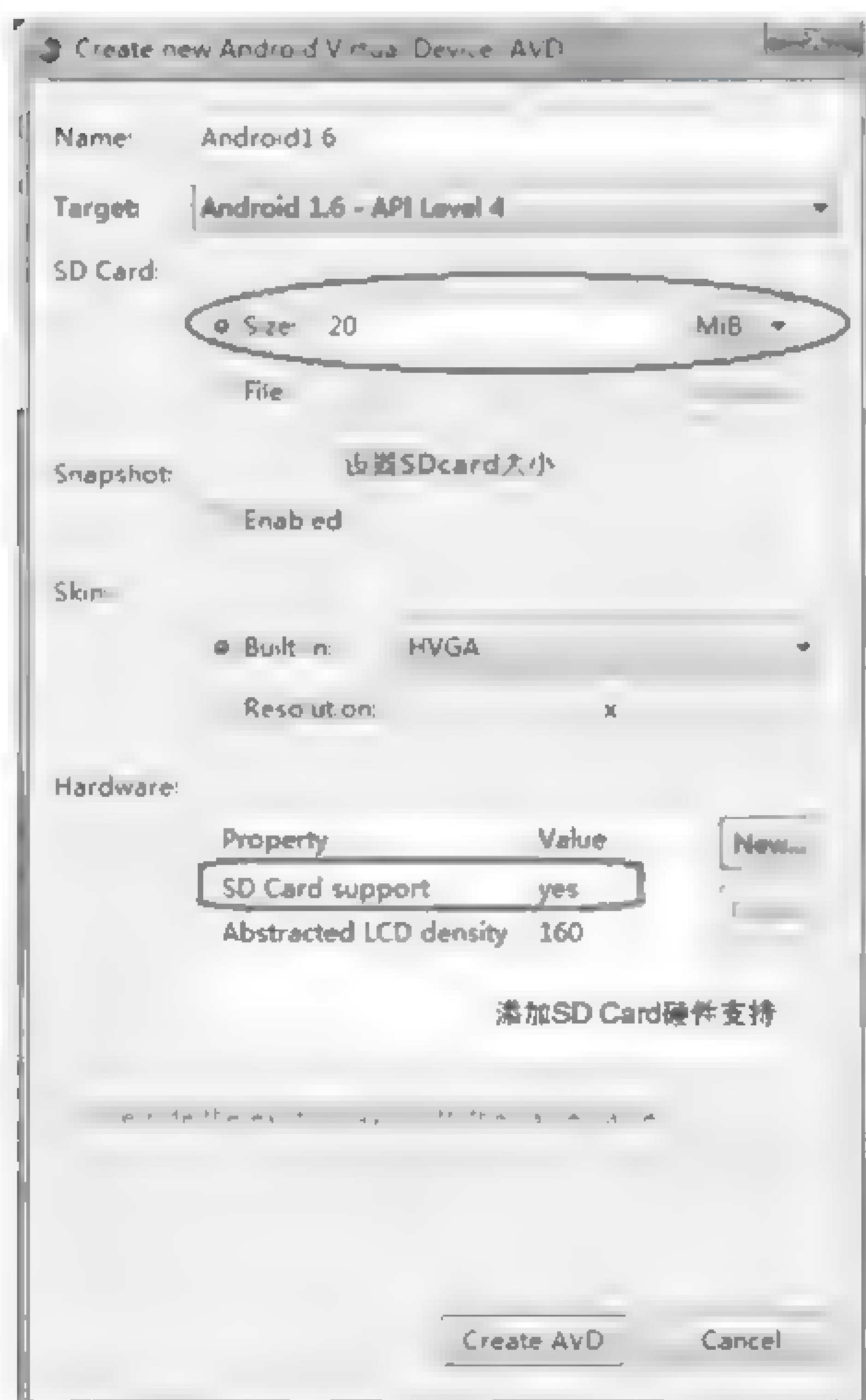


图 2-41 创建 SDcard

- 输入 SDcard 大小（至少要 9M），这里是以 MB 为单位。
- 在“Hardware”列表框下单击“New”按钮，添加“SD Card support”——SD Card 硬件支持。

2.7.9 模拟器横竖屏切换

模拟器横竖屏切换的方法是同时按下电脑键盘的“Ctrl”和“F11”两个按键，或者同时按下“Ctrl”和“F12”也可以。

2.7.10 打包 Android 项目

所有的 Android 应用程序都以“.apk”为后缀命名一个安装包，这个 apk 包中包含应用编译后源码、资源文件、应用的版本信息、用到的权限等等。

在 Android 系统的手机上安装 apk 文件的时候，其应用程序不仅用到“.apk”为后缀的文件，而且还必须利用数字证书签名后才可安装到手机或

者模拟器上。

所谓给程序“签名”，其实就是标识应用程序的作者和应用程序之间建立信任关系，是一种维护知识产权的方式；当然数字证书也不需要权威认证，它是应用程序自我认证的。

那么说到这里，大家会有疑问：为什么在 Eclipse 中运行项目的时候，模拟器和真机都可以正常调试和运行呢？其实当运行一个 Android 项目的时候，内置的 ADT 插件会用内置的调试证书给程序进行打包签名，然后再安装到真机或者模拟器上运行；但是如果想将程序正式发布的话，就不能使用调试的证书签名，必须用正式的签名才可以。

1．Android 程序打包和签名

下面讲解如何将 Android 程序打包并对其签名。

在 Eclipse 中鼠标右键单击需要打包的 Android 项目，然后选中“Android Tools”级联菜单，如图 2-42 所示。

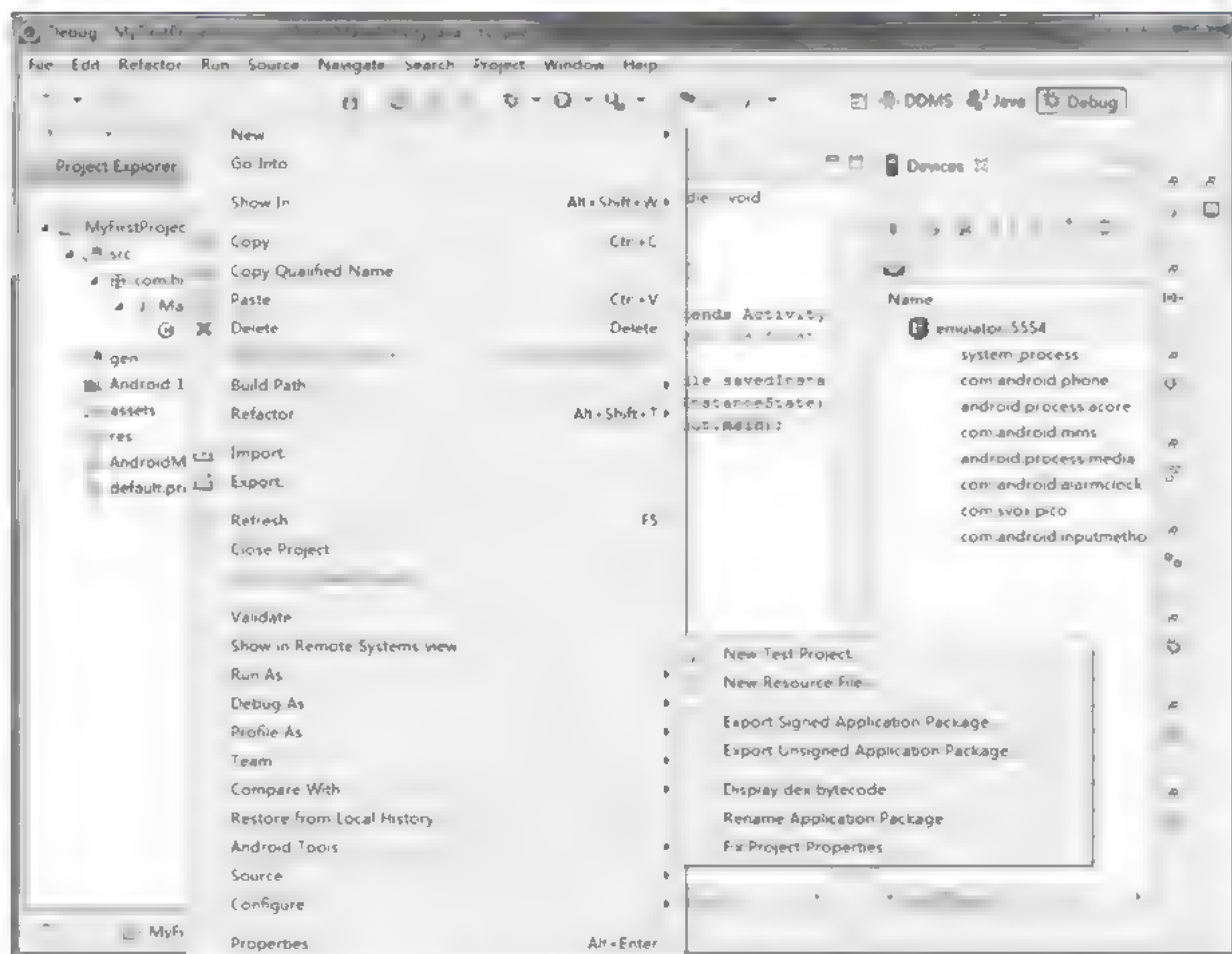


图 2-42 选择打包步骤

在如图 2-42 所示的菜单中，“Export Sigend Application Package...”选项表示签名打包；“Export Unsigned Application Package...”选项表示不签名打包。两种签名方式的具体步骤说明如下：

（1）签名打包

单击“Export Signed Application Package...”选项进入签名打包的第一个窗口“Export Android Application”，如图 2-43 所示。



图 2-43　签名打包—选择打包项目

单击“Browse...”按钮选择打包的项目，然后单击“Next”按钮进入如图 2-44 所示的界面。



图 2-44　签名打包—新建数字证书

在图 2-44 所示的界面中，选中“Create new keystore”单选按钮（新建一个数字证书），输入证书名称和密码，然后单击“Next”按钮进入设置证书信息界面，如图 2-45 所示。



图 2-45 签名打包——证书信息设置

填写好证书信息之后，单击“Next”按钮出现签名打包的最后一个界面，如图 2-46 所示。

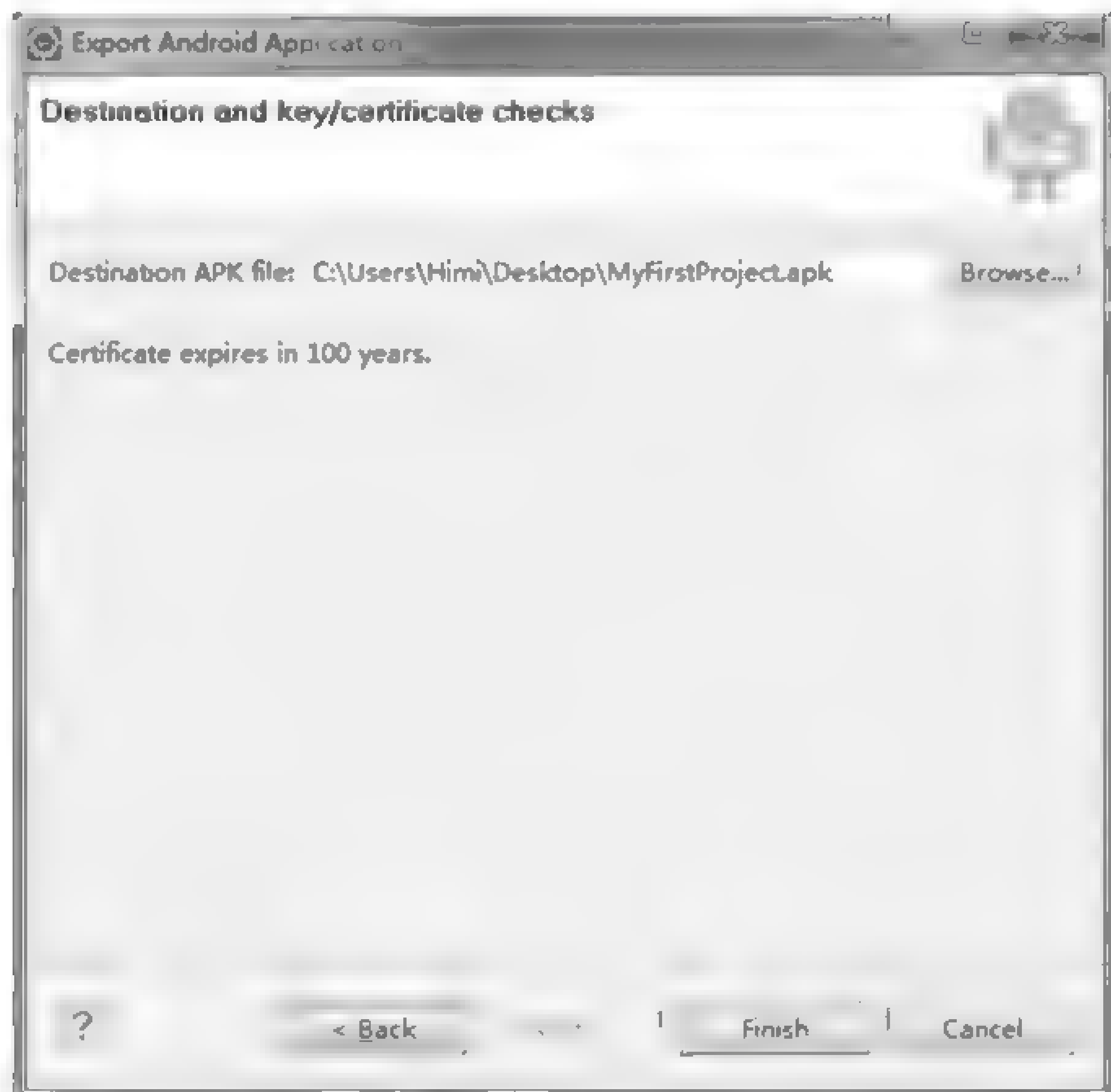
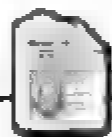


图 2-46 签名打包——选择导出的 apk 包路径

（2）非签名打包

在如图 2-42 所示的级联菜单上，选中“Export Unsigned Application Package…”选项，然后直接选择打包后 apk 的保存路径即可。



小技巧

有些时候需要将自己的程序在真机上安装或者给朋友拿去测试，试玩一下，难道只能通过签名打包才行么？

不用。虽然程序在非签名打包时无法在真机和模拟器上运行，但是，在讲解两种打包之前曾经解释过，当 Android 项目每次运行时，ADT 其实都会为程序利用调试签名进行打包，也就是说只要项目运行过，就会有一个 apk 生成了；这个由 ADT 调试签名的 apk 其实就在项目下的 bin 文件夹里，bin 文件夹在 Eclipse 的项目结构中是看不到的，可以直接到文件系统中查找，例如 MyFirstProject 项目中由 ADT 调试签名的 apk 路径为“eclipse_workspace（Eclipse 的工作路径）\MyFirstProject\bin\MyFirstProject.apk”，这个 APK 虽然不能正式发布出去，但是使用很方便。

2. 安装、卸载 APK 文件

这里先介绍一下 ADB（Android Debug Bridge）工具。ADB 是 Android SDK 自带的工具，使用这个工具可以直接操作管理 Android 模拟器或者真实的 Andriod 设备。

借助 ADB 工具，可以安装和卸载程序，将 Android 设备的文件导出到 PC 上，也可以将 PC 文件导入 Android 设备中等等。简单地说，ADB 其实就是 Android 设备与 PC 之间的一个桥梁，通过 ADB 可以更全面地对 Android 设备进行操作。

那么，如何使用 ADB 命令来进行安装和卸载 APK 程序呢？

首先，打开 DOS 窗口，将目录转到 SDK 中的 adb.exe 所在的目录中。例如，作者电脑的 adb.exe 当前路径是 D:\android-sdk-windows\tools，如图 2-47 所示。

```
D:\android-sdk-windows\tools>adb install d:\test.apk
765 KB/s (13328 bytes in 0.017s)
        pkg: /data/local/tmp/test.apk
Success
```

图 2-47　作者电脑的 adb.exe 当前路径

然后假设“D:\”根目录下有一个“test.apk”文件，我们使用 ADB 命令来对其进行安装和卸载操作，在安装或删除程序之前要确定当前是否有一个模拟器在运行。

安装命令为 adb install [apk 路径]，如图 2-48 所示。

```
D:\android-sdk-windows\tools>adb install d:\test.apk
765 KB/s (13328 bytes in 0.017s)
        pkg: /data/local/tmp/test.apk
Success
```

图 2-48　安装命令

卸载命令为 adb unstall [程序的包路径]，如图 2-49 所示。

```
D:\android-sdk-windows\tools>adb uninstall com.himi
Success
```

图 2-49 卸载命令

需要注意，卸载程序时使用的程序名不是安装时 apk 的文件名，而是程序的包名。

每次想删除和安装一个 apk 程序的时候都要去先转到 ADB 所在路径很麻烦，其实只须将 ADB 所在路径配置到电脑的环境变量中即可。

2.8 本章小结

本章介绍如何创建 Android 项目，熟悉 Android 项目结构，详细剖析 Android 开发中最重要的 Activity 的生命周期以及常见的问题解答等。

本章的重点是 Activity 的生命周期。作为一个 Android 开发者，不仅仅需要会使用 Android API 来进行应用的开发，更应该将 Activity 的生命周期融会贯通。

第 3 章

Android 游戏开发常用的系统控件

从本章节可以学习到：

- ❖ Button
- ❖ Layout
- ❖ ImageButton
- ❖ EditText
- ❖ CheckBox
- ❖ RadioButton
- ❖ ProgressBar
- ❖ SeekBar
- ❖ TabSpec 与 TabHost
- ❖ ListView
- ❖ Dialog
- ❖ 系统控件常见问题



手机应用开发中，不论基于哪个平台，软件基本都是利用系统控件（组件）做开发，虽然系统组件很多，但游戏开发中很少用到；在游戏开发中一般自定义（代码实现）符合游戏题材的组件；因为游戏除了丰富的玩法之外，最注重的就是界面（UI），如果也都利用系统组件做开发，那游戏就会趋于单调，让玩家感到审美疲惫；但是在开发中我们也难免会去跟系统组件打交道，所以这一章向大家讲解一些比较基本、常用的系统组件。

这里需要提醒读者，为了简化篇幅、便于讲解，新建一个项目时都不会去截图和讲述是如何创建该项目的；所以在这里定下一条规则：只要是新建项目，如没有特殊说明的都默认活动名（application name）为“MainActivity”。

3.1 Button

Button（按钮）是一个常用的系统小组件，很小但是在开发中最常用到。一般通过与监听器搭配使用，从而触发一些特定事件。下面来新建一个项目“ButtonProject”，对应的源代码为“3-1（Button 与点击监听器）”。

在上一章中已经介绍过，当新建一个 Android 项目时，ADT 会自动生成一个文本视图（TextView）组件，并显示在屏幕上，那么现在再来添加一个 Button。

打开“res-layout”下的 main.xml 与 string.xml，修改代码如下所示：

main.xml：

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
      <TextView
          android:layout_width="fill_parent"
          android:layout_height="wrap_content"
          android:text="@string/hello"
          />
      <Button
          android:layout_width="fill_parent"
          android:layout_height="wrap_content"
          android:text="确定"
          />
      <Button
          android:layout_width="wrap_content"
          android:layout_height="fill_parent"
          android:text="@string/btn_cancel"
```

```
        />
</LinearLayout>
```

string.xml：

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
   <string name="hello">Hello World, MainActivity!</string>
   <string name="app_name">ButtonProject</string>
   <string name="btn_cancel">取消</string>
</resources>
```

这里新添加了两个 Button，且定义了每个 Button 的宽高和文本属性，但是这两个 Button 是有区别的，其中定义两个 Button 的宽高样式正好相反。“wrap_content”表示是自适应文本宽度样式，“fill_parent”是全部填充样式，可能这么说不太容易理解，在下文看到运行效果后再继续讲解。

另外一点区别就是“android：text”这个文本属性的赋值书写形式不一样。其实这里想说明的是，字符串的定义不是一定要到 string.xml 文件中定义，而后利用“@string/”形式去索引 string.xml 中定义的字符串变量并对其进行赋值；字符串也可以直接在定义组件属性时进行赋值。

String 标签的其他地方都不需要修改，查看一下运行效果，如图 3-1 所示。



图 3-1　两个 Button

在图 3-1 所示的界面中 ，Android 在解析布局文件的时候是从上往下来进行的，也就是说先定义的组件将先被显示出来；那么在布局文件中，可以看到定义组件的顺序依次为“TextView”、按钮（确定）、按钮（取消），所以屏幕上先显示的是新建项目自动生成的“TextView”文本组件，接着是定义的“确定”与“取消”按钮组件。

“确定”按钮在定义的时候，宽为“fill_parent”填充样式，所以它的宽占了整个屏幕，

高定义为“wrap_content”自适应文本样式，所以其高度正好包裹了“确定”字体；“取消”按钮的宽高定义的样式正好与“确定”按钮相反，所以其宽度正好是包裹在“取消”字体；高度则是填充整个屏幕，但是因为“确定”按钮显示的时候已经占了一定高度，所以“取消”按钮的高度没有占满屏幕。

上面添加的两个按钮虽然可以点击，但是没有任何的效果。下面对两个按钮添加点击事件，使“确定”按钮点击后改变“TextView”的文本信息为“确定按钮触发事件！”，并且点击“取消”按钮后改变“TextView”的文本信息为“取消按钮触发事件！”。

步骤1 在布局中给每个组件都新声明 ID 属性。

当需要在源代码中获取到在布局中定义的这 3 个组件时，要给 3 个组件添加 ID 属性。main.xml 代码修改如下：

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
      <TextView
          android:layout_width="fill_parent"
          android:layout_height="wrap_content"
          android:text="@string/hello"
          android:id="@+id/tv"
          />
      <Button
          android:layout_width="fill_parent"
          android:layout_height="wrap_content"
          android:text="确定"
          android:id="@+id/btn_ok"
          />
      <Button
          android:layout_width="wrap_content"
          android:layout_height="fill_parent"
          android:text="@string/btn_cancel"
          android:id="@+id/btn_cancel"
          />
</LinearLayout>
```

给组件添加 ID 属性：定义格式为 android：id=“@+id/name”，这里的 name 是自定义的，不是索引变量。“@+”表示新声明，“@”则表示引用，例如：

- “@+id/tv”表示新声明一个 id，是 id 名为 tv 的组件；
- “@id/tv”表示引用 id 名为 tv 的组件。

为每个组件添加好新声明的 ID 之后，在源代码中通过 R 资源就可以引用到，然后修改 MainActivity.java 文件，源代码如下：

```
package com.himi.button;//包路径
//import 导入类库
import android.app.Activity;
import android.os.Bundle;
import android.widget.Button;
import android.widget.TextView;
public class MainActivity extends Activity {
      private Button btn_ok, btn_cancel;//声明两个按钮对象
      private TextView tv; //声明文本视图对象
      @Override
      public void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.main);
            //对 btn_ok 对象进行实例化
            btn_ok = (Button) findViewById(R.id.btn_ok);
            //对 btn_cancel 对象进行实例化
            btn_cancel = (Button) findViewById(R.id.btn_cancel);
            //对 tv 对象进行实例化
            tv = (TextView) findViewById(R.id.tv);
      }
}
```

关于导入类库有 3 种方式：

- 手动 import，例如声明一个 Button，手动"import android.widger.Button"；
- 声明类型的时候，由 Eclipse 自动补全，并自动导入包。例如声明一个 Button，写 Button 类型的时候写成"Butto" 这里故意将 Button 这个类型单词不写全，然后利用键盘组合键"Alt+/" 自动完成；
- 使用导包快捷键：利用组合键"Shift+Ctrl+O"快速完成导入。先声明两个 Button 与一个 TextView 对象，然后通过 findViewByld（int id）的方法引用在布局文件中定义的组件并且对其进行实例化。

步骤2 给按钮添加点击事件响应。

想知道按钮是否被用户点击，就需要一个"点击监听器"来对其进行监听，然后通过监听器来获取点击的事件，就可以判定关注的按钮是否被用户所点击。

使用监听器有两种方式：

（1）当前类使用点击监听器接口，修改后源代码如下：

```
//使用点击监听器
public class MainActivity extends Activity implements OnClickListener{
```

```
    private Button btn_ok, btn_cancel;
    private TextView tv;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        btn_ok = (Button) findViewById(R.id.btn_ok);
        btn_cancel = (Button) findViewById(R.id.btn_cancel);
        tv = (TextView) findViewById(R.id.tv);
        //将btn_ok按钮绑定在点击监听器上
        btn_ok.setOnClickListener(this);
        //将btn_cancel按钮绑定在点击监听器上
        btn_cancel.setOnClickListener(this);
    }
    //使用点击监听器必须重写其抽象函数,
    //@Override 表示重写函数
    @Override
    public void onClick(View v) {
         if(v == btn_ok){
              tv.setText("确定按钮触发事件！");
         }else if(v == btn_cancel){
              tv.setText("取消按钮触发事件！");
         }
    }
}
```

先用当前类使用点击监听器接口（onClickListener），重写点击监听器的抽象函数（onClick）；然后对需要监听的按钮进行按钮绑定监听器操作，这样监听器才能对绑定的按钮进行监听，以判断其是否被用户点击，一旦有按钮被点击，就会自动响应 onClick 函数，并将点击的 button（button 也是 1 个 view）传入；最后就可以在 onClick 函数中书写点击会触发的事件（因为定义了多个按钮，所以在 onClick 函数中对系统传入的 view 进行按钮匹配的判断，让不同的按钮做不同的处理事件）。

（2）使用内部类实现点击监听器进行监听，修改后源代码如下：

```
public class MainActivity extends Activity{
    private Button btn_ok, btn_cancel;
    private TextView tv;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        btn_ok = (Button) findViewById(R.id.btn_ok);
        btn_cancel = (Button) findViewById(R.id.btn_cancel);
        tv = (TextView) findViewById(R.id.tv);
        //将btn_ok按钮绑定点击监听器
```

```
            btn_ok.setOnClickListener(new OnClickListener() {
                @Override
                public void onClick(View arg0) {
                    tv.setText("确定按钮触发事件！");
                }
            });
            //将 btn_cancel 按钮绑定点击监听器
            btn_cancel.setOnClickListener(new OnClickListener() {
                @Override
                public void onClick(View arg0) {
                    tv.setText("取消按钮触发事件！");
                }
            });
    }
}
```

利用内部类的形式也需要重写点击监听器的抽象函数，然后在 onClick 里进行处理事件，这里不用判断 view 了，因为一个 Button 对应了一个监听器。

3.2 Layout

这里首先开始讲解布局管理的原因在于，系统控件一般都会搭载进布局中，让 Layout 进行管理和规整组件的位置，然后只要需通过 Activity 去显示一个布局，那么布局中搭载的组件就都一并显示在手机屏幕上了。Android 提供了 5 种布局类型，通过这 5 种布局之间的相互组合可以构建各种复杂的布局，五种布局对应的源代码为"3-2（Layout 布局）"。

3.2.1 线性布局

LinearLayout（线性布局）是 5 种布局中最常用的一种，此布局在显示组件的时候会默认保持组件之间的间隔以及组件之间的互相对齐（相对一个组件的右对齐、中间对齐或者左对齐）。线性布局显示组件的方式有垂直与水平两种，可以通过 orientation 进行设定。

新建一个项目，然后在布局文件（main.xml）中添加 3 个名字不同但其他属性都相同的按钮组件。

修改后的 main.xml：

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
```

```
    android:layout_height="fill_parent"
    >
      <Button
          android:layout_width="fill_parent"
          android:layout_height="wrap_content"
          android:text="Button1"
          />
      <Button
          android:layout_width="fill_parent"
          android:layout_height="wrap_content"
          android:text="Button2"
          />
      <Button
          android:layout_width="fill_parent"
          android:layout_height="wrap_content"
          android:text="Button3"
          />
</LinearLayout>
```

orientation 属性表示设置布局中的控件的方向，其属性值有两种，一种是“vertical”垂直排列，另一种是“horizontal”水平排列。这里设置成了垂直排列，宽高都设置成了填充的形式。运行项目后的效果如图 3-2 所示。

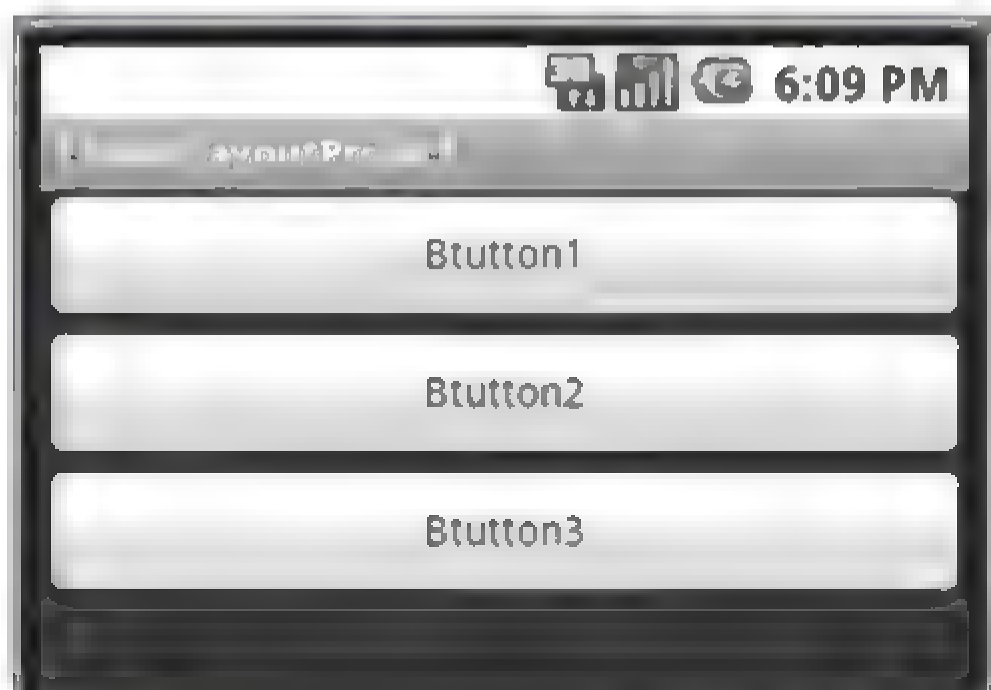


图 3-2　线性布局方式—垂直显示组件

在图 3-2 中可以看到 3 个按钮组件都依次垂直排列的，那么如果将布局的 orientation 属性设置成水平方式，项目运行效果如图 3-3 所示。



图 3-3　线性布局方式-水平显示组件

这里只有 1 个 Button 显示在屏幕中，其实剩下的 2 个按钮组件不是没有显示，而是显示在了 Button1 的右侧，因为定义的每个按钮组件的宽都是填充形式，所以第 1 个按钮的宽横向填充屏幕，剩余的两个则超出屏幕导致无法看到。一开始就说过了，LinearLayout 布局就是将组件从左往右放置；而图 3-2 能显示全部的 3 个按钮组件，是因为将 LinearLayout 的显示组件的方向设置成了垂直放置。

为了让大家看的更清楚，下面将定义的 3 个 Button 组件的宽属性改成自适应内容的形式，然后查看运行效果，如图 3-4 所示。



图 3-4　线性布局方式-组件宽度自适应

设置了每个按钮的宽之后，可以在屏幕中完整的看到 3 个按钮组件，这也证实了图 3-3 中只显示一个按钮的原因。

下面在 LinearLayout 里继续嵌套一个 LinearLayout 布局，然后修改 main.xml 如下：

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
      <Button
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:text="Button1"
          />
    <LinearLayout android:orientation="vertical"
          android:layout_width="fill_parent"
          android:layout_height="fill_parent"
          >
            <Button
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="Button2"
          />
            <Button
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
```

```
                android:text="Button3"
                />
     </LinearLayout>
</LinearLayout>
```

修改后的布局文件将 Button2 与 Button3 放在了嵌套的 LinearLayout 当中，并且嵌套的 LinearLayout 属性与最外层的 LinearLayout 宽高属性定义一样，只是嵌套的布局将控件显示方向设置成了垂直方向，那么来看运行的效果（如图 3-5 所示）：

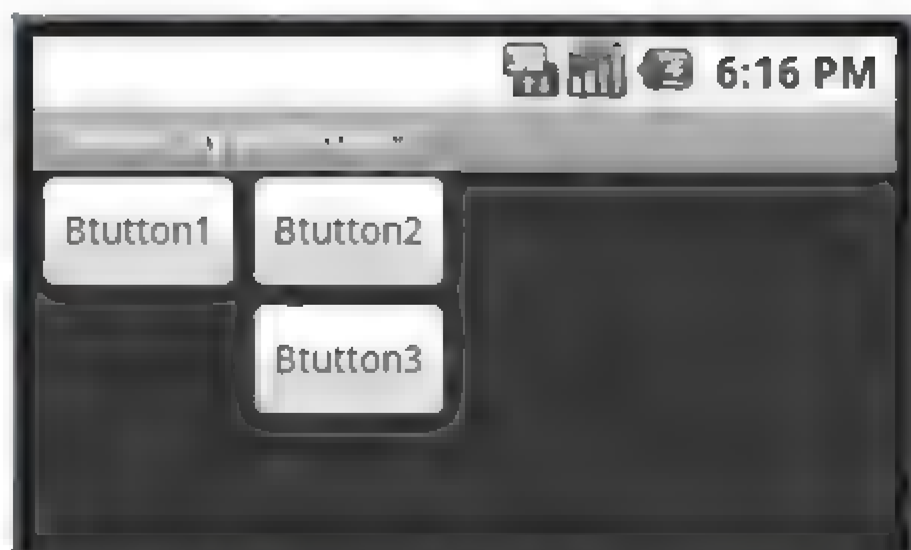


图 3-5　线性布局方式-嵌套布局

在图 3-5 所示的界面中，可以看到 Button2 与 Button3 垂直放置，这是因为嵌套的线性布局的影响；而 Button1 与 Button2、Button3 仍然是以水平方式排列，这是因为最外层的线性布局的影响。

下面来介绍一个重要的属性“Layout_weight”。所有的组件都有 Layout_weight 属性，不设置的情况下，默认为零。其属性表示当前还有多大视图就占据多大的视图；如果其值高于零，则表示将父视图中可用的空间进行分割，分割的大小视当前屏幕整体布局的 Layout_weight 值与每个组件 Layout_weight 值的占用比例而定。

修改布局代码如下:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
      <Button
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:text="Button1"
          />
      <LinearLayout android:orientation="vertical"
          android:layout_width="fill_parent"
          android:layout_height="fill_parent"
          >
             <Button
```

```
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="Button2"
                android:layout_weight="1"
            />
            <Button
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="Button3"
                android:layout_weight="1"
                />
    </LinearLayout>
</LinearLayout>
```

在 Button2 与 Button3 组件里多定义了 android:layout_weight 属性，然后看一下运行后的效果，如图 3-6 所示。

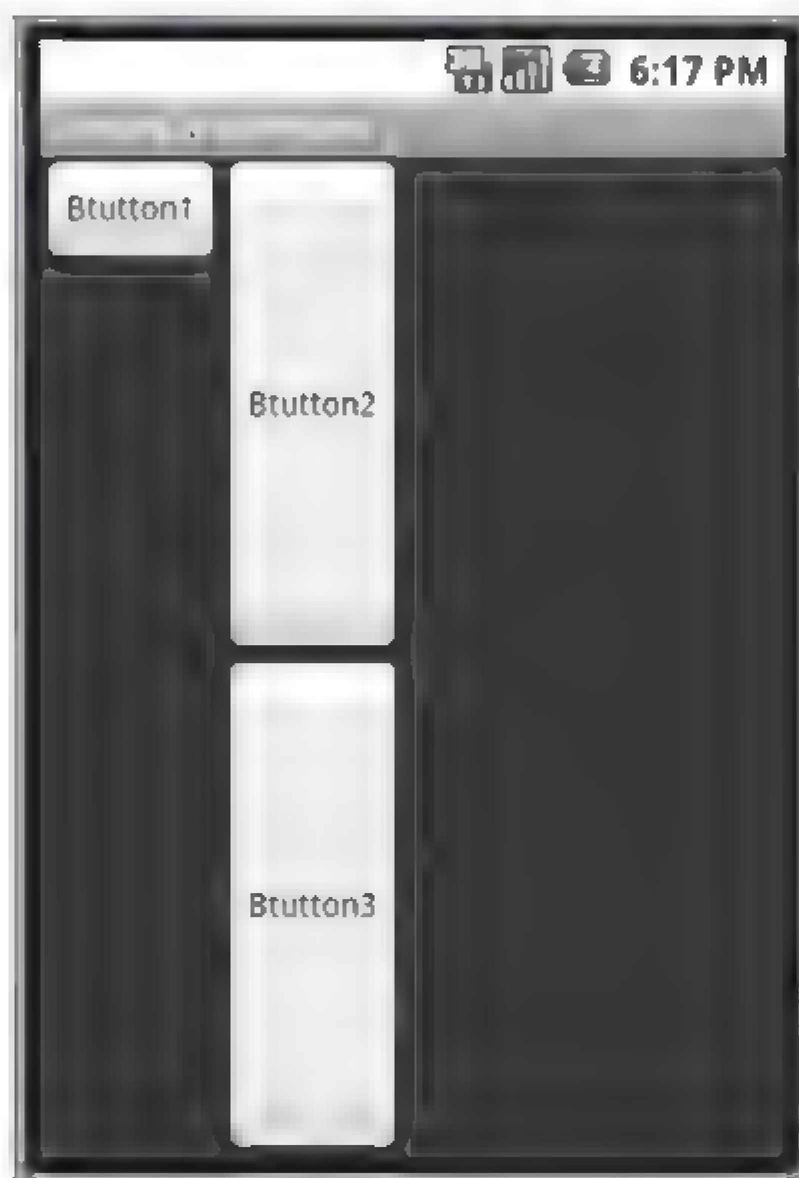


图 3-6　线性布局方式-layout_weight 属性

因为 Button2 与 Button3 添加了 android:layout_weight 属性，而且其值为 1，大于零了，所以将嵌套 Button2 与 Button3 的 LinearLayout 布局剩余空间全部占据。又因为 Button2 与 Button3 的 android:layout_weight 属性值都是 1，所以空间被两者平分占据；而横向没有占据的原因是因为嵌套 Button2 与 Button3 的 LinearLayout 布局，设置了显示组件方式为垂直显示，所以 Button2 与 Button3 的父视图剩余空间只有纵向空间。

线性布局中常用的属性还有 3 种：

（1）gravity ：每个组件默认其值为左上角对齐，其属性可以调整组件对齐方式比如向左、向右、或者居中对齐等。

（2）padding: 边距的填充，也称内边距。其边距属性有：

android:paddingTop，设置上边距；

android:padding Bottom，设置下边距；

android:paddingLeft，设置左边距；

android:paddingRight，设置右边距；

android:padding 则表示周围四方向各内边距统一调整。

边距属性值为具体数字。

（3）layout_margin：外边距，其上下左右属性为：

android:layout_ marginTop，设置上边距；

android:layout_marginBottom，设置下边距；

android:layout_marginLeft，设置左边距；

android:layout_marginRight，设置右边距；

android:layout_margin 则表示设置四方向边距统一调整。

padding 与 layout_margin 的区别如图 3-7 所示。

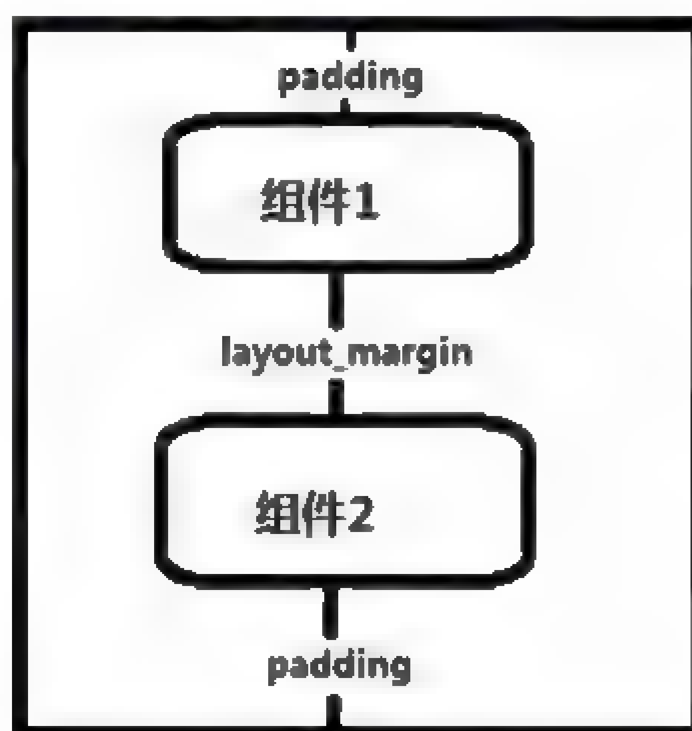


图 3-7　两种边距的区别

padding 内边距指的是当前布局与包含的组件之间的边距；

layout_margin 外边距指的是与其他组件之间的边距。

3.2.2　相对布局

RelativeLayout（相对布局）：除了最常用的 LinearLayout 之外，相对布局则是另外一种常用的布局。与线性布局不同之处在于，线性布局如果需要将一组件对齐另一个组件就必须将所有的组件进行对齐，或者使用嵌套布局才可完成；但是相对布局不必这么麻烦。因为在相对布局中，每个组件都可以指定相对于其它组件或父组件的位置，只是必须通过 ID 来进行指定。修改 main.xml 布局如下：

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
```

```
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
      <Button
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:text="Button1"
          android:id="@+id/btn1"
          />
      <Button
          android:layout_width="fill_parent"
          android:layout_height="wrap_content"
          android:text="Button2"
          android:id="@+id/btn2"
          android:layout_below="@id/btn1"
          />
      <Button
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:text="Button3"
          android:id="@+id/btn3"
          android:layout_below="@id/btn2"
          android:layout_alignRight="@id/btn2"
          />
      <Button
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:text="Button4"
          android:id="@+id/btn4"
          android:layout_below="@id/btn3"
          android:layout_alignParentRight="true"
          />
      <Button
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:text="Button5"
          android:id="@+id/btn5"
          android:layout_below="@id/btn4"
          android:layout_centerHorizontal="true"
          />
</RelativeLayout>
```

先来看运行的效果，如图 3-8 所示。

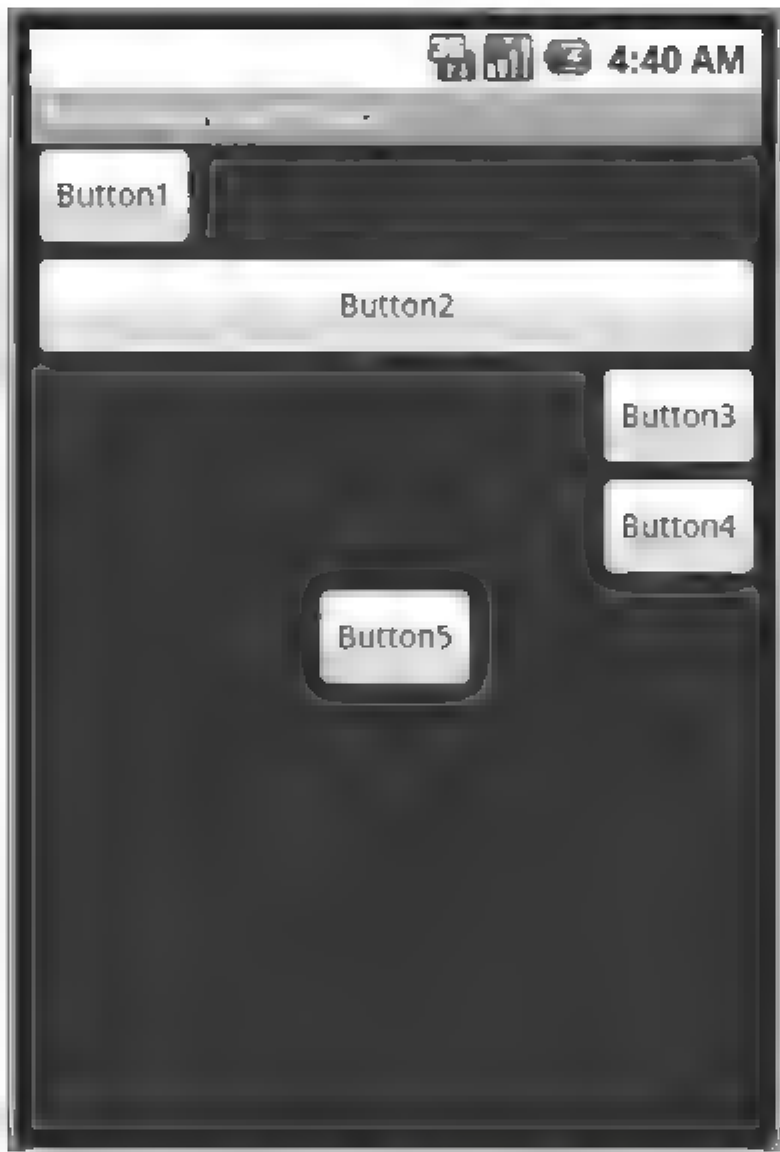


图 3-8 相对布局

在布局文件中使用相对布局并定义了 5 个 Button，下面对每个 Button 属性的设置进行详细讲解。

①Button1：设置了宽、高、ID 和按钮文本。

②Button2：设置了组件之间位置关系的属性。

android:layout_below="@id/btn1" 表示该组件位置放在 ID 为 btn1 组件的下方，除此之外组件之间的位置关系还有 3 个属性，详细如表 3-1 所示（属性值为组件 ID）。

表 3-1 组件之间的位置关系

属性名称	作用
android：layout_above	将该组件放在指定 ID 组件的上方
android：layout_below	将该组件放在指定 ID 组件的下方
android：layout_toLeftOf	将该组件放在指定 ID 组件的左方
android：layout_toRightOf	将该组件放在指定 ID 组件的右方

③Button3：设置了组件之间对齐方式的属性。

android:layout_alignRight="@id/btn2"表示 Button3 与 ID 为 btn2 的组件进行右边缘对齐；其他几种对齐方式如表 3-2 所示（属性值为组件 ID）。

④Button4：设置了组件与父组件之间的对齐方式的属性。

android:layout_alignParentRight="true"表示当前组件与父组件进行右边缘对齐；其他几种与父组件对齐方式如表 3-3 所示（属性值只有 true 和 false，默认为 false）。

表 3-2 组件对齐方式

属性名称	作用
android：layout alignBaseline	将该组件放在指定 ID 组件进行中心线对齐
android：layout alignTop	将该组件放在指定 ID 组件进行顶部对齐
android：layout alignBottom	将该组件放在指定 ID 组件进行底部对齐
android：layout alignLeft	将该组件放在指定 ID 组件进行左边缘对齐
android：layout alignRight	将该组件放在指定 ID 组件进行右边缘对齐

表 3-3 当前组件与父组件的对齐方式

属性名称	作用
android：layout_alignParentTop	该组件与父组件进行顶部对齐
android：layout_alignParentBottom	该组件与父组件进行底部对齐
android：layout_alignParentLeft	该组件与父组件进行左边缘对齐
android：layout_alignParentRight	该组件与父组件进行右边缘对齐

⑤Button5:设置了组件方向的属性。

android:layout_centerHorizontal ="true"表示该组件放置在水平方向的中央位置；其他方向的属性如表 3-4 所示（属性值只有 true 和 false，默认为 false）。

表 3-4 组件放置的位置

属性名称	作用
android：layout_centerHorizontal	将该组件放置在水平方向中央的位置
android：layout_centerVertical	将该组件放放置在垂直方向的中央位置
android：layout_centerInParent	将该组件放置在父组件的水平中央及垂直中央的位置

在相对布局中，设置一个组件的位置，一般要有“上下”与“左右”两个位置属性来进行固定，当然有时用一个也是对的。但是设置一种就可固定其组件位置的原因在于 Android 在排版组件的时候，如不设置位置等属性，默认仍在屏幕最上角，也就是说使用一个位置属性能正常固定组件所放的位置，是因为系统默认有一个“最左”和“最上”的隐藏位置属性。

例如：有 3 个按钮 button1、button 2、button 3，布局定义如下：

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout width="fill parent"
    android:layout height="fill parent"
    >
```

```
    <Button
        android:text="Button1"
        android:id="@+id/button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content">
        </Button>
    <Button
        android:text="Button2"
        android:id="@+id/button2"
        android:layout_toRightOf="@id/button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content">
        </Button>
    <Button
        android:text="Button3"
        android:id="@+id/button3"
        android:layout_below="@id/button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content">
        </Button>
</RelativeLayout>
```

在布局中，定义了3个按钮组件，Button1默认放在屏幕最上角，Button2放在Button1的右侧，Button3放在Button1的下方，运行效果如图3-9所示。



图3-9　三个按钮的布局

不再修改这3个按钮的属性，然后添加一个Button4，其Button4在布局中的定义如下：

```
<Button
    android:text="Button4"
    android:layout_toRightOf="@id/button3"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content">
</Button>
```

很明显，将会认为Button4放在了Button3的右侧，那么查看一下此时的运行效果，如图3-10所示。



图 3-10　四个按钮的布局

但是，Button4 的位置不是理想中放在 Button3 的右侧，而是覆盖掉了 Button2。其实也不奇怪，因为只是设定了 Button4 放在 Button3 的右侧，上下位置并没有设置，并且 Button2 也在 Button3 的右侧，所以 Button4 被放置在了和 Button2 一样的位置上，这也证实了一个位置属性是无法固定组件的。

那么为什么 Button2，Button3 也是一个位置属性，就可以正常显示呢？

Button2 能正常放在 Button1 的右侧，那是因为 Button2 的上方没有组件存在，如果还有组件存在，那么 Button2 的上下位置肯定也是错的！这证实了隐藏属性“最上”。

Button3 能正常显示在 Button1 下方也是因为 Button3 的左侧没有其他组件存在，如果其左侧还有组件，那么 Button3 的位置也肯定是错的！这证实了隐藏属性“最左”。

所以在相对布局中，如果像固定一个组件的位置，至少要确定组件 “左右”与“上下”两个位置才可准确固定组件位置。

3.2.3　表格布局

TableLayout（表格布局）的样式如同一个表格。通常情况下，TableLayout 有多个 TableRow 组成，每个 TableRow 就是一行，定义几个 TableRow 就是定义几行；TableLayout 不会显示行列号，也没有分割线。其行数和列数也是由自己来操作和确定的。

下面先看一个简单的表格布局的效果，如图 3-11 所示。



图 3-11　表格布局

再来看 main.xml 中如何实现的：

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<TableLayout xmlns:android "http://schemas.android.com/apk/res/android"
    android:layout_width-"fill_parent"
    android:layout_height="fill_parent"
    >
    <TableRow>
      <Button android:text="Button1" />
            <Button android:text="Button2" />
            <Button android:text="Button3"/>
    </TableRow>
    <TableRow>
            <Button android:text="Button4" />
            <Button android:text="Button5" />
            <Button android:text="Button6" />
    </TableRow>
    <TableRow>
            <Button android:text="Button7" />
            <Button android:text="Button8" />
            <Button android:text="Button9" />
    </TableRow>
</TableLayout>
```

布局文件很简单，在表格布局中定义了3行，每一行中有3个Button组件。

下面来介绍在TableLayout中常使用的几个属性：

（1）shrinkColumns属性：以0为序，当TableRow里面的控件布满布局时，指定列自动延伸以填充可用部分；当TableRow里面的控件还没有布满布局时，shrinkColumns不起作用。

在布局中添加shrinkColumns属性代码如下：

```
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:shrinkColumns="2"
>
```

添加设置表格布局的android:shrinkColumns="2"，指定第3列填充可用部分，然后运行项目，效果如图3-12所示。

虽然没有任何的改变，但是这也证实了，当TableRow里的控件还没布满布局的时候，shrinkColumns属性不起作用，因为Button3右侧还有不少空间；那么，来加长Button3这个按钮的名称使其填满TableRow布局，运行项目后的效果如图3-13所示。

不设置shrinkColumns属性之前的原效果如图3-14所示。通过这两个图的对比，就很清晰shrinkColumns的作用了。



图 3-12　android:shrinkColumns="2"的布局



图 3-13　加长 Button3 按钮名称的布局



图 3-14　不设置 shrinkColumns 属性之前的原效果

（2）strechColumns 属性：以第 0 行为序，指定列对空白部分进行填充。

在布局中添加 strechColumns 属性代码如下：

```
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:stretchColumns="2"
>
```

添加设置表格布局的 android:stretchColumns="2"，指定第 3 列填充空白部分，然后运行项目，效果如图 3-15 所示。

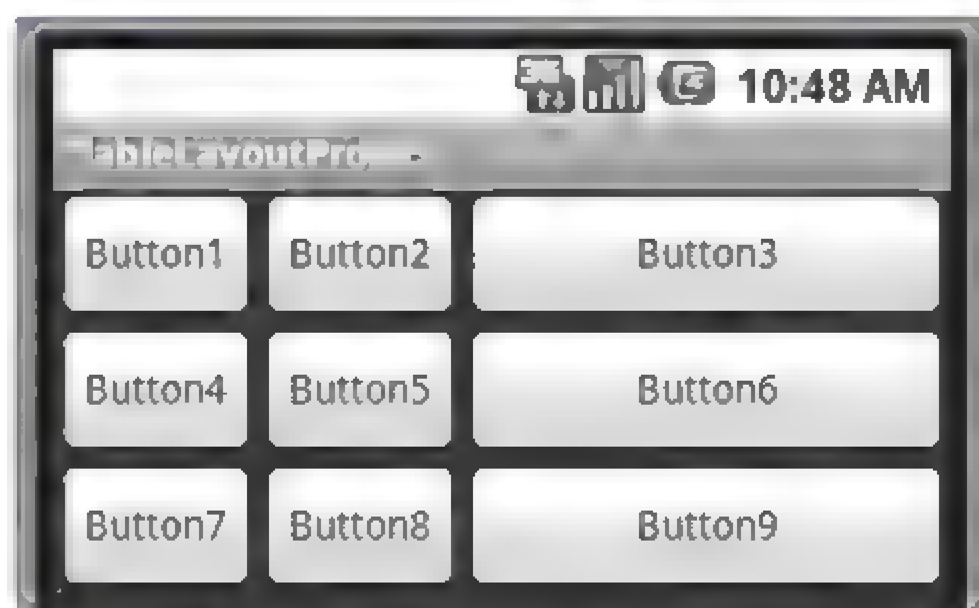


图 3-15　设置 android:stretchColumns="2"的布局

可以明显看到3×3的按钮矩阵剩余的空间被第3列自动填充了。

（3）collapseColumns 属性：以第0行为序，隐藏指定的列。

在布局中添加 collapseColumns 属性代码如下：

```
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:collapseColumns="2"
>
```

添加设置表格布局的 android:collapseColumns="2"，指定第 3 列隐藏，然后运行项目，效果如图 3-16 所示。



图 3-16　设置 android:collapseColumns="2"的布局

可以明显看到3×3的按钮矩阵第3列被隐藏了。

（4）layout_column 属性：以第0行为序，设置组件显示在指定列。

（5）layout_span 属性：以第0行为序，设置组件显示占用的列数。

修改布局代码如下：

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
    <TableRow>
      <Button android:text="Button1" android:layout_span="3"/>
            <Button android:text="Button2" />
            <Button android:text="Button3" />
    </TableRow>
    <TableRow>
            <Button android:text="Button4" android:layout_column="2" />
            <Button android:text="Button5" android:layout_column="0" />
            <Button android:text="Button6" />
    </TableRow>
    <TableRow>
```

```
            <Button android:text="Button7" />
            <Button android:text="Button8" />
            <Button android:text="Button9" />
    </TableRow>
</TableLayout>
```

Button1 设置显示占用 3 个列数大小的空间， Button4 设置显示在第 3 列，Button2 设置显示在第 4 列。运行项目，然后查看运行效果，如图 3-17 所示。



图 3-17　设置 android:layout_span="3"的布局

从上面效果图中可以看到，Button1 确实占用了 3 个列数大小的空间，Button4 也通过设置正常显示在了第 3 列，但是 Button5 没有按照设定的显示在第 1 列，原因在于在表格布局中，TableRow 一行里的组件都会自动放在前一组件的右侧，依次排列。所以只要 TableRow 行中第一个组件确定了所在列，其后者就无法再次进行位置设定；其实当 Button4 的设置起了作用，同时也已经影响到了 Button5 和 Button6 的位置。

3.2.4　绝对布局

AbsoluteLayout（绝对布局）布局用法如其名，组件的位置可以准确的指定其在屏幕的 x/y 坐标位置。虽然可以精确的去规定坐标，但是由于代码的书写过于刚硬，使得在不同的设备，不同分辨率的手机移动设备上不能很好的显示应有的效果，所以此布局不推荐使用。

修改布局文件如下：

```
<?xml version="1.0" encoding="utf-8"?>
<AbsoluteLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
      <Button
          android:layout_width="wrap_content"
          android:layout_height="fill_parent"
          android:text="Button1"
          />
```

```
    <Button
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Button2"
        />
</AbsoluteLayout>
```

布局使用绝对布局，并且定义 2 个 Button 组件，运行项目查看效果，如图 3-18 所示。



图 3-18 绝对布局

从上面运行的效果图中，可以看到组件之间没有自动对齐、没有间隔，都默认放在屏幕的最左上角。如果想改变组件的位置，只能通过设置具体的 x/y 坐标，利用属性 layout_x 与 layout_y 进行设置。

添加 layout_x 与 layout_y 属性，修改布局代码如下：

```
<?xml version="1.0" encoding="utf-8"?>
<AbsoluteLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
    <Button
        android:layout_width="wrap_content"
        android:layout_height="fill_parent"
        android:text="Button1"
        android:layout_x="100dp"
        />
    <Button
```

```
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Button2"
        android:layout_y="100dp"
        />
</AbsoluteLayout>
```

Button1 的 x 坐标设定为了 100 像素，Button2 的 y 坐标设定成了 100 像素。来看运行效果，如图 3-19 所示。

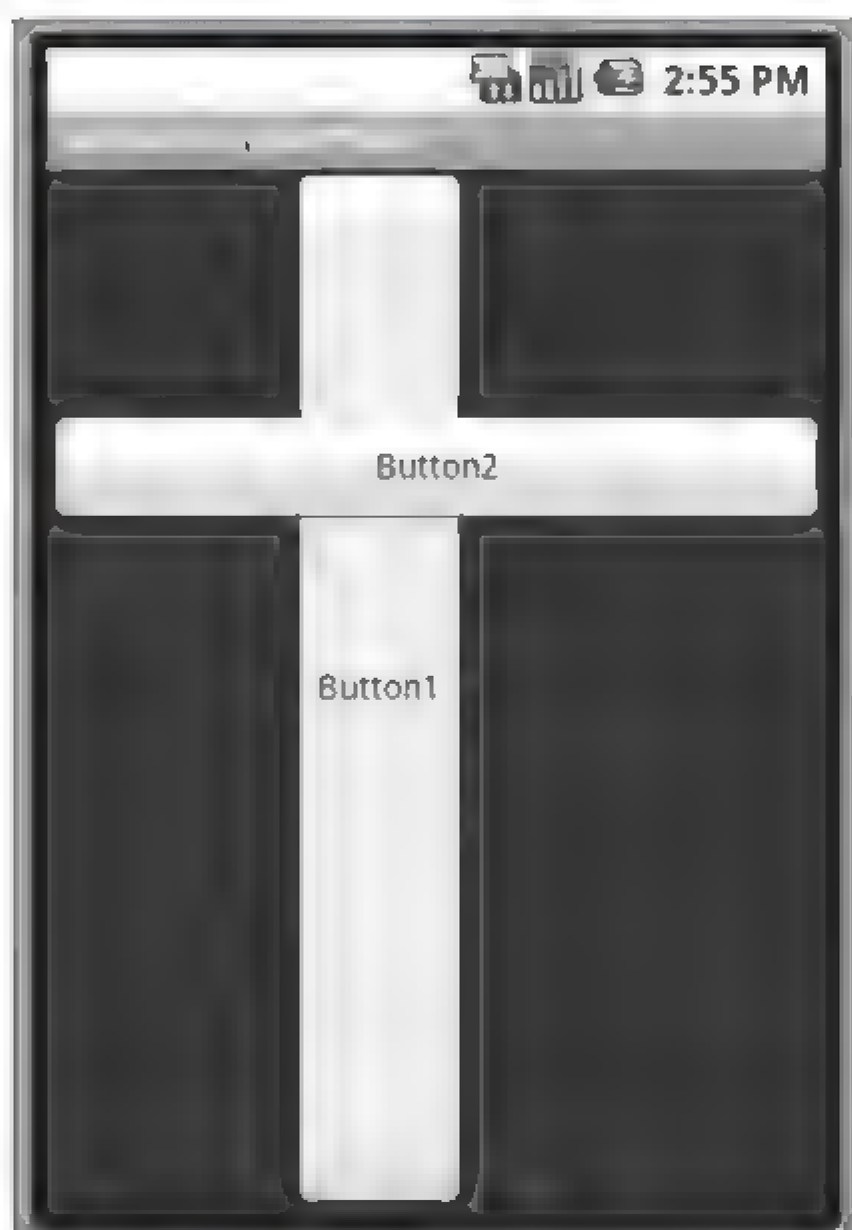


图 3-19　绝对布局设置组件位置

此布局设定组件位置都使用 x/y 坐标来进行调整，使得整体布局格式代码很不灵活，所以再次提醒大家不到不得已的时候，尽可能不要去沾染此布局。

3.2.5　单帧布局

FrameLayout（单帧布局）是 5 种布局中最简单的一种，因为单帧布局在新定义组件的时候永远都会将组件放在屏幕的左上角，即使在此布局中定义多个组件，后一个组件总会将前一个组件所覆盖，除非最后一个组件是透明的。

修改布局代码如下：

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
```

```
    >
      <Button
          android:layout_width="fill_parent"
          android:layout_height="wrap_content"
          android:text="Button1"
          />
      <Button
          android:layout_width="wrap_content"
          android:layout_height="fill_parent"
          android:text="Button2"
      />
      <Button
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:text="Button3"
          />
</FrameLayout>
```

在单帧布局中仍是定义了 3 个按钮组件，只是每个按钮的宽高属性不同，然后查看运行效果，如图 3-20 所示。



图 3-20　单帧布局

从图 3-20 中可以看到定义的 3 个按钮组件都有重叠部分，单帧布局不会像线性布局那样每个组件之间自动对齐并且组件之间都有间隔，所以在单帧布局中定义 3 个按钮的时候故意将每个按钮的组件宽高设定不同值。如果定义成一样的话，就只能在屏幕中看到 Button3 这个组件，因为其他的 2 个组件都将会被 Button3 组件所覆盖。

3.2.6 可视化编写布局

在编写布局文件窗口的最左下方，可以看到一个“Graphical Layout”页面窗口，如图 3-21 所示。



图 3-21　布局编写窗口

单击进入布局可视化编辑窗口，如图 3-22 所示。

左侧都是一些常用组件、布局类型等等，只要选中需要的拖动到右侧的模拟手机屏幕的窗口中，按照自己的意图去编放位置，在 main.xml 中即可自动生成对应的布局代码。

虽然可视化很方便，但是建议大家尽可能不要去使用，因为通过可视化去实现的布局，即使能编出自己满意的布局，可对其布局的框架结构能了解多少呢？不出问题没什么区别，一旦出现问题，找问题又成了难点。但是如果布局是用代码编写出来的，那就明显会不一样。因为使用代码编写的过程中，布局的整体结构和层次都已经在脑海中形成，而且代码看起来也会很清晰，即便出现问题，找问题也是很容易的。除此之外，并不是所有的布局用此可视工具都能实现。



图 3-22　布局可视化编辑窗口

所以建议大家，尤其对于新手而言，刚接触就一定要多动手编写代码，加深对布局的印象和理解。

3.3 ImageButton

ImageButton 与 Button 类似，区别在于 ImageButton 可以自定义一张图片做为一个按钮；也正因为使用图片替代了按钮，所以 ImageButton 按下与抬起的样式效果需要自定义。下面看看系统按钮组件单击前后的对比图，如图 3-23 所示。



图 3-23　按钮单击前后对比图

对于 ImageButton，这里只讲解如何实现按下的状态切换，由于监听事件与 Button 无区别，不再赘述。

运行项目，效果如图 3-24 所示。

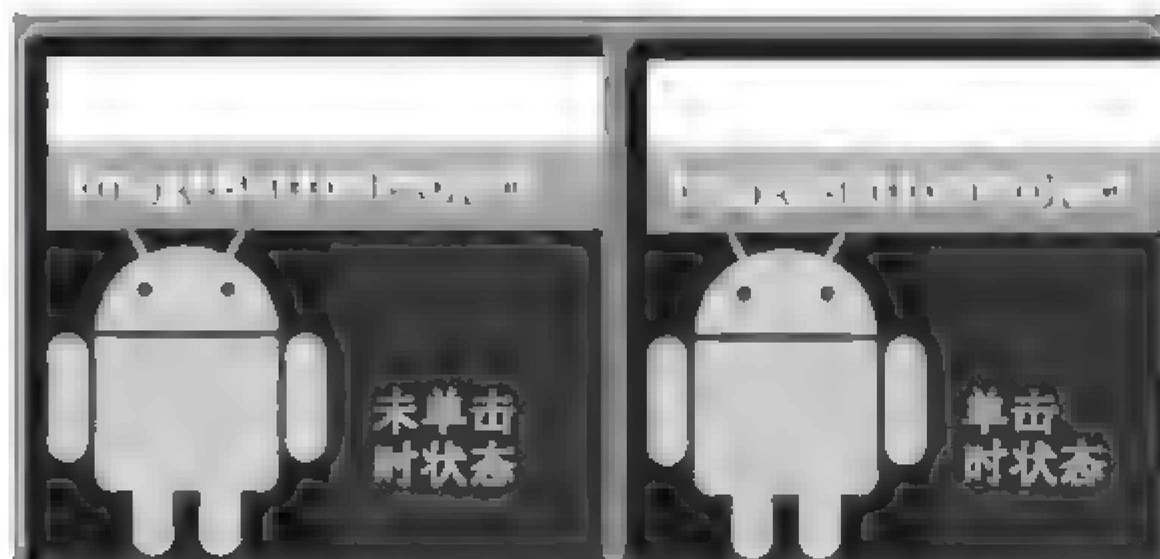


图 3-24　自定义图片按钮单击前后对比图

新建项目“ImageButtonProject”对应的源代码为“3-3（ImageButton 图片按钮）”。在项目中添加两张图片资源，如图 3-25 所示。

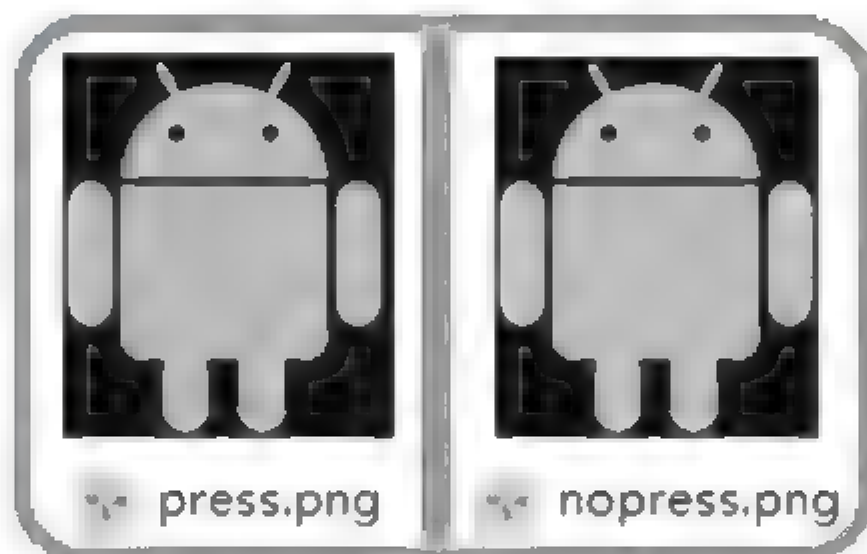


图 3-25　在项目中添加两张图片资源

修改布局代码如下：

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
      <ImageButton
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:id="@+id/imageBtn"
          android:background="@drawable/nopress"
      />
</LinearLayout>
```

以上布局中定义了一个 ImageButton，并为其设定了背景图与 ID，背景图使用未单击的图片。android:background 属性表示为其组件设置背景图，其属性值为索引图片 ID。

因为按钮按下和提起是触屏事件，所以需修改源代码去监听图片按钮触屏事件，在 ImageButton 按下时设置改变背景图即可，修改源代码如下：

```
public class MainActivity extends Activity {
    private ImageButton Ibtn;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        Ibtn = (ImageButton)findViewById(R.id.imageBtn);
        //为图片按钮添加触屏监听
        Ibtn.setOnTouchListener(new OnTouchListener() {
            @Override
            public boolean onTouch(View v, MotionEvent event) {
                    //用户当前为按下
                    if(event.getAction()==MotionEvent.ACTION_DOWN){
                        //设置图片按钮背景图

      Ibtn.setBackgroundDrawable(getResources().getDrawable(
R.drawable.press));
                    //用户当前为抬起
                    }else if(event.getAction()==MotionEvent.ACTION_UP){

      Ibtn.setBackgroundDrawable(getResources().getDrawable(R.drawable.
nopress));
                        }
                        return false;
                    }
            });
    }
}
```

为图片按钮添加设置按下效果图片的具体步骤如下：

步骤1 利用内部类形式完成 ImageButton 绑定触屏监听，这里使用的监听器为 OnTouchListener（触屏监听器），根据此监听器可以得到屏幕是否被用户按下和抬起等触屏事件，这里使用按钮进行绑定则表示用户是否按下按钮、按钮抬起等事件。

步骤2 重写接口中的 onTouch（View v，MotionEvent event）抽象函数。onTouch（View v，MotionEvent event）的参数说明如下：

- 第一个参数：表示触发触屏事件的事件源 view;
- 第二个参数：表示触屏事件的类型，如按下，抬起，移动等。

步骤3 利用 MotionEvent.getAction()函数判断用户触发事件的类型。触发事件有两种类型：

- MotionEvent.ACTION_DOWN：按下事件;
- MotionEvent.ACTION_UP：抬起事件。

步骤4 根据按下与抬起事件的不同，调用 ImageButton 类中的 setBackgroundDrawable()函

数设置 ImageButton 背景图即可。

getResources().getDrawable(int ID)：传入图片 ID 得到一个 Drawable 对象。有关 getResources 在本章 3.10.3 小节中有讲解，不再赘述。

这里对 ImageButton 的按下与抬起事件都做了处理，其原因是当用户抬起按钮事件时能让图片按钮回复到没有点击的状态图，否则将一直处于按下按钮的状态图。

3.4 EditText

EditText（输入框）是与用户交互数据的常用组件，例如在登录游戏，输入帐号、密码等信息时经常用到。

新建项目“EditTextProject”，对应的源代码为“3-4（EditText 文本编辑）”。修改项目的布局代码如下：

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
      <TextView
          android:layout_width="fill_parent"
          android:layout_height="wrap_content"
          android:text="@string/hello"
          android:id="@+id/tv"
          />
     <EditText
          android:layout_width="fill_parent"
          android:layout_height="wrap_content"
             android:hint="提示信息"
             android:id="@+id/et"
       />
      <Button
          android:layout_width="fill_parent"
          android:layout_height="wrap_content"
          android:text="获取EditText 内容!"
          android:id="@+id/btn"
          />
</LinearLayout>
```

这里讲下 EditText 中的 android:hint 属性。hint 是指输入框中没有任何内容的情况下，默

认出现的提示信息；一旦输入框有内容，那么 hint 提示信息将被内容替代。

运行效果如图 3-26 所示。



图 3-26　EditTextProject 项目运行效果图

EditText 中的“提示信息”为 hint 属性的提示内容，而不是 EditText 的文本内容，所以 EditText 中不输入内容时，即使单击按钮获取 EditText 文本内容，也不会获取到“提示信息”字样。

EditText 组件最重要的就是获取用户输入的内容，下面来做一个“单击按钮获取用户在 EditText 中输入的内容”的项目，修改源代码 MainActivity.java 如下：

```
public class MainActivity extends Activity implements OnClickListener{
    private EditText et;//创建一个文本编辑的对象
    private Button btn;
    private TextView tv;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        et= (EditText)findViewById(R.id.et);//实例化文本编辑
        btn= (Button)findViewById(R.id.btn);
        btn.setOnClickListener(this);
        tv = (TextView)findViewById(R.id.tv);
    }
    @Override
    public void onClick(View v) {
        if(v==btn){
            //获取 EditText 中的文本内容
            String str = et.getText().toString();
            //让 TextView 将获取到的 EditText 内容 str 显示出来
            tv.setText(str);
        }
    }
}
```

在 EditText 中输入“himi”，然后单击按钮，效果如图 3-27 所示。



图 3-27　EditText—单击按钮效果图

在没有单击按钮之前，EditText 中显示的为“提示信息”；当输入内容的时候，提示信息消失，单击按钮之后，TextView 中显示了 EditText 中的内容“himi”。

有些时候用户可能会通过 EditText 输入密码，这时为了保护用户隐私，当用户在 EditText 中输入内容的时候，其内容都会被“*”符号所替代；其实这个功能也是 EditText 的一种属性，只需要在布局中定义 EditText 时设置其属性即可。

在布局中添加 EditText 密码输入属性，android:password，设置其值为 true（其默认值为 false），然后重新运行项目，输入内容，效果如图 3-28 所示。



图 3-28　EditText 内容输入为密码形式

EditText 其他常用属性：

- android:numeric="integer"，表示只能在 EditText 中输入数字；
- android:singleLine="true"，表示在 EditText 中输入的内容单行显示，不自动换行；
- android:maxLength="1"，表示设置 EditText 输入内容最大长度为 1。

3.5 CheckBox

打开项目“CheckBoxProject”，对应的源代码为“3-5（CheckBox 与监听）”，在布局中注册 CheckBox 组件，代码如下：

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
```

```
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
       <TextView
          android:layout_width="fill_parent"
          android:layout_height="wrap_content"
          android:text="CheckBoxProject"
          />
       <CheckBox
          android:layout_width="fill_parent"
          android:layout_height="wrap_content"
          android:text="CheckBox1"
          android:id="@+id/cb1"
          />
       <CheckBox
          android:layout_width="fill_parent"
          android:layout_height="wrap_content"
          android:text="CheckBox2"
          android:id="@+id/cb2"
          />
       <CheckBox
          android:layout_width="fill_parent"
          android:layout_height="wrap_content"
          android:text="CheckBox3"
          android:id="@+id/cb3"
           />
</LinearLayout>
```

定义 3 个 CheckBox，然后修改源代码 MainActivity 对 3 个 CheckBox 进行监听。

```
//使用状态改变检查监听器
public class MainActivity extends Activity
                                 implements OnCheckedChangeListener{
      private CheckBox cb1, cb2, cb3;//创建 3 个 CheckBox 对象
      @Override
      public void onCreate(Bundle savedInstanceState) {
             super.onCreate(savedInstanceState);
             setContentView(R.layout.main);
             //实例化 3 个 CheckBox
             cb1 = (CheckBox) findViewById(R.id.cb1);
             cb2 = (CheckBox) findViewById(R.id.cb2);
             cb3 = (CheckBox) findViewById(R.id.cb3);
             cb1.setOnCheckedChangeListener(this);
             cb2.setOnCheckedChangeListener(this);
             cb3.setOnCheckedChangeListener(this);
      }
      //重写监听器的抽象函数
```

```
    @Override
    public void onCheckedChanged(CompoundButton buttonView, boolean
isChecked) {
            //buttonView 选中状态发生改变的那个按钮
            //isChecked 表示按钮新的状态 (true/false)
            if(cb1 == buttonView||cb2 == buttonView ||cb3 == buttonView) {
                    if(isChecked){
                            //显示一个提示信息
                            toastDisplay(buttonView.getText() + "选中");

                    }else{
                            toastDisplay(buttonView.getText() + "取消选中");
                    }
            }
    }
    public void toastDisplay(String str){
            Toast.makeText(this, str, Toast.LENGTH_SHORT).show();
    }
}
```

对 CheckBox 进行监听，步骤如下：

步骤1 使用 OnCheckedChangeListener 接口，这里的接口导入的是:

"android.widget.CompoundButton.OnCheckedChangeListener"；

而不是"android.widget.RadioGroup.OnCheckedChangeListener"。

RadioGroup 下的 OnCheckedChangeListener 接口是对 RadioButton 按钮进行监听的接口。这里千万不要使用错了接口，关于 RadioButton 后文将详细讲解。

步骤2 重写监听器的抽象函数"onCheckedChanged()"。

步骤3 将每个 CheckBox 组件绑定监听器。

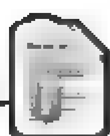
通过重写的 onCheckedChanged（CompoundButton buttonView，boolean isChecked）函数第一个参数来确定哪个 CheckBox 状态发生改变；根据第二个参数来确定改变的 CheckBox 的具体状态值，true 为勾选，false 为未勾选。

MainActivity 类中还定义了 toastDisplay()函数，其实只是为了使用 Android 的一种提示信息的方式：Toast：主要用于提示信息，使用起来很方便；先创建 Toast 对象，然后调用 makeText()方法得到一个 Toast 实例对象。

M makeText（Context context，CharSequence text，int duration）

第一个参数是上下文对象；第二个参数显示的文本内容；第三个参数显示提示消息的持续时间；其值有两个常量：LENGTH_SHORT（短暂持续）和 LENGTH_LONG（略长持续）。

最后，使用 Toast 对象调用 show()方法即可。



提示

如果想让 Toast 提示在一个特定的条件下立即消失，那么首先应该定义一个 Toast 成员变量，然后通过 makeText 函数获取其实例赋值与定义的 Toast 成员变量，这样一来如果想让提示立刻消失，直接让定义的 Toast 成员变量设置为 null 即可。

"CheckBoxProject"项目运行结果如图 3-29 所示。



图 3-29　CheckBoxProject 项目运行效果图

3.6 RadioButton

RadioButton 是单选按钮，而上面介绍的 CheckBox 是复选框。RadioButton 一般都存在单选组（RadioGroup）中，当多个 RadioButton 存放在同一个单选组（RadioGroup）中，只能单选一个 RadioButton；如果想实现 RadioButton 的多选，那就需要多个 RadioGroup。

新建项目"RadioButtonProject"，对应的源代码为"3-6（RadioButton 与监听）"。"RadioButtonProject"项目运行效果如图 3-30 所示。



图 3-30　RadioButtonProject 项目运行效果图

修改布局文件：

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
    <RadioGroup
      android:id="@+id/radGrp"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content">
            <RadioButton
                android:layout_width="fill_parent"
                android:layout_height="wrap_content"
                android:text="RadioButton1"
                />
            <RadioButton
                android:layout_width="fill_parent"
                android:layout_height="wrap_content"
                android:text="RadioButton2"
                />
            <RadioButton
                android:layout_width="fill_parent"
                android:layout_height="wrap_content"
```

```
            android:text "RadioButton3"
            />
    </RadioGroup>
</LinearLayout>
```

定义了 3 个 RadioButton，而且都定义在 RadioGroup（单选按钮组）中，绑定监听，修改源代码 MainActivity 如下：

```
//使用状态改变监听器
public class MainActivity extends Activity implements
OnCheckedChangeListener {
    private RadioButton rb1, rb2, rb3;
    private RadioGroup rg;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        rb1 = (RadioButton) findViewById(R.id.rb1);
        rb2 = (RadioButton) findViewById(R.id.rb2);
        rb3 = (RadioButton) findViewById(R.id.rb3);
        rg = (RadioGroup) findViewById(R.id.radGrp);
        rg.setOnCheckedChangeListener(this);//将单选组绑定监听器
    }

    //重写监听器函数
    @Override
    public void onCheckedChanged(RadioGroup group, int checkedId) {
        if(group==rg){//因为当前程序中只有一个RadioGroup，此步可以不进行判定
            String rbName = null;
            if (checkedId == rb1.getId()) {
                rbName = rb1.getText().toString();
            } else if (checkedId == rb2.getId()) {
                rbName = rb2.getText().toString();
            } else if (checkedId == rb3.getId()) {
                rbName = rb3.getText().toString();
            }
            Toast.makeText(this,"选择了下标为"" + rbName+""的单选按钮",
                    Toast.LENGTH_LONG).show();
        }
    }
}
```

RadioButton 与 CheckBox 监听步骤类似，但 RadioButton 监听需要注意三点：

- RadioButton 与 CheckBox 使用的监听器不同。

- RadioButton 绑定监听的时候，不是每个 RadioButton 都去绑定，因为所有的 RadioButton 都被放在了 RadioGroup 单选组中，所以只需要将 RadioGroup 绑定上监听器即可。
- 重写监听器函数 onCheckedChanged（RadioGroup group，**int** checkedId），这个函数的第一个参数是单选组，注意第二个参数，这里的 checkedId 不是 RadioGroup 单选组中每个 RadioButton 的下标，而是发生状态改变的 RadioButton 的内存 ID！这一点一定要注意。所以在进行判断哪个 RadioButton 发生状态改变的时候，可以利用 RadioButton.getID 来与 checkedId 进行对比。

3.7 ProgressBar

在 Android 应用开发中，ProgressBar（运行进度条）是较常用到的组件，例如下载进度、安装程序进度、加载资源进度显示条等。在 Android 中提供了两种样式来分别表示在不同状态下显示的进度条，下面来实现这两种样式。

新建项目“ProgressBarProject”，源代码为“3-7（ProgressBar 进度条）”，修改布局文件如下，运行效果如图 3-31 所示。



图 3-31　进度条运行效果图

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
        <TextView
             android:layout_width="wrap_content"
             android:layout_height="wrap_content"
```

```
            android:text="默认进度条: "
            />
        <ProgressBar
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="progress1"
            />
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="小圆形进度条: "
            />
        <ProgressBar
             android:layout_width="wrap_content"
             android:layout_height="wrap_content"
             android:text="progress3"
             style="?android:attr/progressBarStyleSmall"
            />
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="大圆形进度条: "
            android:layout_gravity="center_vertical"
            />
        <ProgressBar
             android:layout_width="wrap_content"
             android:layout_height="wrap_content"
             android:text="progress3"
             style="?android:attr/progressBarStyleLarge"
            />
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="条形进度条: "
            />
        <ProgressBar
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:text="progress2"
            android:id="@+id/pb"
            style="?android:attr/progressBarStyleHorizontal"
            android:max="100"
            android:progress="50"
            android:secondaryProgress="70"
            />
</LinearLayout>
```

默认进度条是圆形，通过 style 属性来指定系统进度条的大小：

- style="?android:attr/progressBarStyleSmall"，小圆形进度条。
- style="?android:attr/progressBarStyleLarge"，大圆形进度条。

如果需要将进度显示为长条形，那么 style 必须设定为这种类型：

- style="?android:attr/ progressBarStyleHorizontal"，长条形进度条。

针对长条形进度条，还有几个常用属性：

- android:max，设置进度条最大进度值。
- android:progress，设置进度条初始进度值。
- android:secondaryProgress，设置底层（浅色）进度值。

圆形显示进度条默认是动态、但是长条进度显示条却是静态的，那么修改源代码 MainActivity 实现长条进度条为动态显示：

```
//使用 Runnable 接口
public class MainActivity extends Activity implements Runnable {
      private Thread th ;              //声明一条线程
      private ProgressBar pb ;  //声明一个进度条对象
      private boolean stateChage; //标识进度值最大最小的状态
      @Override
      public void onCreate(Bundle savedInstanceState) {
             super.onCreate(savedInstanceState);
             setContentView(R.layout.main);
             //实例进度条对象
             pb = (ProgressBar)findViewById(R.id.porb);
             th = new Thread(this);//实例线程对象
             th.start();//启动线程
      }

      @Override
      public void run() {//实现 Runnable 接口抽象函数
             while(true){
                    int current=pb.getProgress();//得到当前进度值
                    int currentMax=pb.getMax();//得到进度条的最大进度值
                    int secCurrent=pb.getSecondaryProgress();//得到底层当前进度值
                    //以下代码实现进度值越来越大，越来越小的一个动态效果
                    if(stateChage==false){
                           if(current>=currentMax){
                                  stateChage true;
                           }else{
                                  //设置进度值
                                  pb.setProgress(current+1);
```

```
                    //设置底层进度值
                    pb.setSecondaryProgress(current+1);
                }
            }else{
                if(current<=0){
                    stateChage=false;
                }else{
                    pb.setProgress(current-1);
                }
            }
            try {
                Thread.sleep(50);
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
    }
}
```

ProgressBar 类中常用函数如下所示：

- getProgress()：获取当前进度值函数;
- setProgress()：设置进度值函数;
- getSecondaryProgress ()：获取底层进度值函数;
- setSecondaryProgress ()：设置底层进度值函数;
- getMax()：获取当前最大进度值函数。

在线程循环中对进度条的最大进度值与当前进度值进行判定处理，然后不断设置进度值进而达到动态进度值越来越大，或越来越小的动态效果。

3.8 SeekBar

SeekBar（拖动条）的外观类似长条进度条。Android 手机上最常见到拖动条的地方就是在播放音乐的时候，当用户在拖动条上任意拖动可以调整音乐播放的时间段；调整铃声音量大小界面也是利用的拖动条与用户进行交互。下面来学习如何定义和监听拖动条事件。

首先在布局文件中注册一个拖动条组件，源代码为“3-8（SeekBar 拖动条）”，运行效果如图 3-32 所示。



图 3-32　SeekProject 项目运行效果图

新建项目“SeekProject”，修改布局代码如下：

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
      <TextView
          android:layout_width="fill_parent"
          android:layout_height="wrap_content"
          android:text="SeekBarProject"
          android:id="@+id/tv"
          />
      <SeekBar
             android:layout_width="fill_parent"
          android:layout_height="wrap_content"
          android:text="SeekBar"
          android:id="@+id/seekbar"
          />
</LinearLayout>
```

上述代码简单声明了 TextView 和 SeekBar 组件；下面对拖动条进行监听，修改源代码 MainActivity.java，修改如下：

```
public class MainActivity extends Activity {
      private SeekBar seekBar;
      private TextView tv;

      @Override
      public void onCreate(Bundle savedInstanceState) {
             super.onCreate(savedInstanceState);
             setContentView(R.layout.main);
             seekBar = (SeekBar) findViewById(R.id.seekbar);
             tv = (TextView) findViewById(R.id.tv);
             seekBar.setOnSeekBarChangeListener(new
                           OnSeekBarChangeListener() {
                    @Override
                    public void onStopTrackingTouch(SeekBar seekBar) {
```

```
                    tv.setText("<拖动条>完成拖动");
                }
                @Override
                public void onStartTrackingTouch(SeekBar seekBar) {
                    tv.setText("<拖动条>拖动中...");
                }
                @Override
                public void onProgressChanged(SeekBar seekBar,int
                        progress, boolean fromUser) {
                    tv.setText("当前<拖动条>的值为: "+progress);
                }
            });
    }
}
```

对拖动条进行监听的是 setOnSeekBarChangeListener 这个接口，这里使用的内部类实现绑定监听，然后重写接口的三个函数：

- onStopTrackingTouch: 当用户对拖动条的拖动动作完成时触发；
- onStartTrackingTouch: 当用户对拖动条进行拖动时触发；
- onProgressChanged: 当拖动条的值发生改变的时触发。

其实拖动条类似长条提示进度条，也拥有 setMax()、setProgress()、setSecondaryProgress () 这些函数。

3.9 TabSpec 与 TabHost

TabHost 相当于浏览器中浏览器分页的集合，而 Tabspec 则相当于浏览器中的每个分页面；在 Android 中，每一个 TabSpec 分页可以是一个组件，也可以是一个布局，然后将每个分页装入 TabHost 中，TabHost 即可将其中的每个分页一并显示出来。

本节范例源代码为“3-9（Tab 分页式菜单）”，运行范例项目，效果如图 3-33 所示。

打开项目“TabProject”，在项目资源中添加了两张图片资源 bg.png 与 bg2.png ，如图 3-34 所示。



图 3-33　分页式布局



图 3-34　项目资源中添加了两张图片资源

接下来修改布局代码如下：

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
     android:background="@drawable/bg2"
    >
      <Button
          android:layout_width="fill_parent"
          android:layout_height="wrap_content"
          android:text="This is Tab1"
          android:id="@+id/btn"
          />
      <EditText
          android:layout_width="fill_parent"
          android:layout_height="wrap_content"
          android:text="This is Tab2"
          android:id="@+id/et"
           />
      <LinearLayout xmlns:android="http://schemas.android.com/apk/res/
                                         android"
          android:orientation="vertical"
```

```
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:id="@+id/mylayout"
        android:background="@drawable/bg"
        >
          <Button
              android:layout_width="fill_parent"
              android:layout_height="wrap_content"
              android:text="This is Tab3"
              />
         <EditText
              android:layout_width="fill_parent"
              android:layout_height="wrap_content"
              android:text="This is Tab3"
                />
    </LinearLayout>
</LinearLayout>
```

布局中定义了按钮和编辑文本组件，只是有两个组件定义在了一个嵌套的布局中；每个布局属性都设置了 background 属性，其含义是为布局添加背景图片，利用“@drawable”来索引资源文件下的图片。

下面来看 MainActivity 中的代码：

```
public class MainActivity extends TabActivity implements
OnTabChangeListener {
     private TabSpec ts1, ts2, ts3;//声明3个分页
     private TabHost tableHost;//分页菜单(tab的容器)
     @Override
     public void onCreate(Bundle savedInstanceState) {
           super.onCreate(savedInstanceState);
           tableHost = this.getTabHost();//实例(分页)菜单
           //利用LayoutInflater将布局与分页菜单一起显示
           LayoutInflater.from(this).inflate(R.layout.main,
                     tableHost.getTabContentView());
           ts1 = tableHost.newTabSpec("tabOne");//实例化一个分页
           ts1.setIndicator("Tab1");//设置此分页显示的标题
           ts1.setContent(R.id.btn);//设置此分页的资源id
           ts2 = tableHost.newTabSpec("tabTwo");
           //设置此分页显示的标题和图标
           ts2.setIndicator("Tab2",getResources().getDrawable(
                     R.drawable.icon));
```

```
            ts2.setContent(R.id.et);
            ts3 = tableHost.newTabSpec("tabThree");
            ts3.setIndicator("Tab3");
            ts3.setContent(R.id.mylayout);//设置此分页的布局id
            tableHost.addTab(ts1);//菜单中添加ts1分页
            tableHost.addTab(ts2);
            tableHost.addTab(ts3);
            tableHost.setOnTabChangedListener(this);
    }
    @Override
    public void onTabChanged(String tabId) {
            if (tabId.equals("tabOne")) {
               Toast.makeText(this, "分页1", Toast.LENGTH_LONG).show();
            }
            if (tabId.equals("tabTwo")) {
               Toast.makeText(this, "分页2", Toast.LENGTH_LONG).show();
            }
            if (tabId.equals("tabThree")) {
               Toast.makeText(this, "分页3", Toast.LENGTH_LONG).show();
            }
    }
}
```

显示分页式布局，详细步骤如下：

（1）继承 TabActivity：在此之前继承的都是 android.app.Activity 类，但是这里需要继承 android.app.TabActivity。

（2）创建 TabHost 分页菜单对象，利用以下代码。

```
LayoutInflater.from(this).inflate(R.layout.main,
                    tableHost.getTabContentView());
```

将 main.xml 布局与 tabHost 一并显示在屏幕中；有关 LayoutInflater 类在后续文章中有详细讲解。

（3）实例声明了 3 个 TabSpec 分页对象，然后添加到 TabHost 中；这里对每一个分页 ts1,ts2,ts3 都设置了标题与资源 ID。

TabSpec 类的 setIndicator（CharSequence arg0）函数，第一个参数表示设置分页标题，第二个参数表示设置分页的图标。

TabSpec 类的 setContent（int arg0）函数，传入的可以是组件 ID，也可以是布局 ID；分

页 ts1 与 ts2 设置的是组件，而 ts3 设置的则是布局；要注意的是这里不管传入的是组件 ID 还是布局 ID，都应该是 main.xml 布局中的 ID；因为在利用 LayoutInflater 显示布局与分页菜单时，这里的参数已经指定 main.xml 的布局 ID 了。

从图 3-33 可以看到第二个分页带有图标，其实就是源代码中的 ts2 分页，因为 ts2 分页使用的是两个参数的 setIndicator 函数，不仅设置了标题还设置了图标。

最后利用 TabHost 类中的 addTab()函数将分页添加进去（往 TabHost 添加的先后顺序就是在屏幕中从左往右的页面顺序）。

下面来监听分页改变事件，具体步骤如下：

步骤1 使用 OnTabChangedListener 接口，重写 OnTabChanged（String tabId）函数。

步骤2 TabHost 绑定监听器。

步骤3 判断 OnTabChanged（String tabId）中的 tabId 参数进行处理事件；这里的 tabId 对应的是实例中每个分页传入的分页 ID，而不是 TabSpec.setIndicator()设置的标题。

3.10 ListView

ListView（列表视图）是一个常用的组件，其数据内容以列表形式直观的展示出来，比如做一个游戏的排行榜，对话列表等等都可以使用列表来实现，且 ListView 的优点是列表中的数据可以自适应屏幕大小。

首先介绍“适配器”这个基础概念。在列表中定义的数据都通过“适配器”来映射到 ListView 上，ListView 中常用的适配器有两种：

- ArrayAdapter：最简单的适配器，只能显示一行文字；
- SimpleAdapter：具有很好扩展性的适配器，可以显示自定义内容。

3.10.1 ListView 使用 ArrayAdapter 适配器

使用 ArrayAdapter 适配器的范例源代码为“3-10-1（列表之 ArrayAdapter 适配）”。打开项目“ListViewProject_1”，修改源代码 MainActivity:

```
public class MainActivity extends Activity {
    private ListView lv ;//声明一个列表
    private List<String> list ;//声明一个 List 容器
    private ArrayAdapter<String> aa ;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
```

```
        setContentView(R.layout.main);
        lv = new ListView(this);//实例化列表
        list = new ArrayList<String>();//实例化 List
        //往容器中添加数据
        list.add("Item1");
        list.add("Item2");
        list.add("Item3");
        //实例适配器
        //第一个参数：Context
        //第二个参数：ListView 中每一行布局样式
        //android.R.layout.simple_list_item_1：系统中每行只显示一行文字布局
        //第三个参数：列表数据容器
        aa = new ArrayAdapter<String>(this,
            android.R.layout.simple_list_item_1,list);
        lv.setAdapter(aa);//将适配器数据映射 ListView 上
        this.setContentView(lv);
    }
}
```

显示一个带有数据的 ListView 的步骤如下：

步骤1 实例一个添加数据的容器，并将数据放入容器。

步骤2 实例列表适配器，并且实例适配器时将数据传入。

步骤3 实例一个 ListView，并且为其设置适配器。

步骤4 利用 setContentView()函数显示 ListView。

运行范例项目，效果如图 3-35 所示。

在项目中，需要为列表添加单击事件监听。让一个列表绑定单击事件监听，只需要将 ListView 设置监听器即可，添加监听事件代码如下：

```
lv.setOnItemClickListener(new OnItemClickListener() {
      @Override
      public void onItemClick(AdapterView<?> arg0, View arg1,
      int arg2, long arg3) {
          Toast.makeText(MainActivity.this, "当前选中列表项的下标为:
              "+arg2, Toast.LENGTH_SHORT).show();
      }
});
```



图 3-35　ListView（ArrayAdapter）

因为列表中每一项数据都是一个 Item，所以将 ListView 绑定使用 OnItemClickListener 项单击监听器，并且重写监听器中的 onItemClick()函数。onItemClick()函数的第一个参数是触发的适配器，第二个参数是触发的视图，第三个参数是适配器中项的位置下标，第四个参数 listView 项下标。

3.10.2　ListView 使用 SimpleAdapter 适配器的扩展列表

使用 SimpleAdapter 适配器的扩展列表的源代码为“3-10-2（列表之 SimpleAdapter 适配）”。

虽然使用 ArrayAdapter 适配器可以显示列表，但是列表的每一项（行）很单调，因为只能显示一行文本很局限。那么想自定义列表每项、自定义布局就要使用 SimpleAdapter 适配器来进行，以对列表进行更好的扩展，例如实现如图 3-36 所示的效果。



图 3-36　ListView（SimpleAdapter）

新建项目“ListViewProject_1”，首先设置 ListView 中每一项的布局，因为要自定义 ListView 中的项，所以不再使用 Android 系统提供的布局。修改 main.xml 如下：

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
<ImageView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/iv"
    />
   <LinearLayout
          android:orientation="vertical"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          >
            <TextView
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:textSize="20sp"
                android:id="@+id/bigtv"
                />
            <TextView
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:textSize="10sp"
                android:id="@+id/smalltv"
                />
    </LinearLayout>
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
     android:text="button"
    android:id="@+id/btn"
    />
<CheckBox
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:id="@+id/cb"
   />
</LinearLayout>
```

首先定义布局为线性布局，设置布局方向为横向水平方式，并在布局中添加了一个ImageView图片组件，一个按钮组件和一个复选框组件。

在添加ImageView组件后还嵌套了一个线性布局，且嵌套布局方向为垂直方向，在嵌套的线性布局中添加了两个TextView组件，这两个TextView组件设置的字体大小不同。

然后修改源代码 MainActivity：

```
public class MainActivity extends Activity {
	private SimpleAdapter sp;//声明适配器对象
	private ListView listView; //声明列表视图对象
	private List<Map<String,Object>> list;//声明列表容器
	@Override
	public void onCreate(Bundle savedInstanceState) {
		super.onCreate(savedInstanceState);
		//实例化列表容器
		list = new ArrayList<Map<String,Object>>();
		listView = new ListView(this);//实例化列表视图
		//实例一个列表数据容器
		Map<String, Object>  map = new HashMap<String, Object>();
		//往列表容器中添加数据
		map.put("item1_imageivew", R.drawable.icon);
		map.put("item1_bigtv", "BIGTV");
		map.put("item1_smalltv", "SMALLTV");
		//将列表数据添加到列表容器中
		list.add(map);
		//实例适配器
		sp = new SimpleAdapter(this, list,R.layout.main, new String[]{
			"item1_imageivew","item1_bigtv", "item1_smalltv"}, new
int[] { R.id.iv,R.id.bigtv, R.id.smalltv});
		//为列表视图设置适配器（将数据映射到列表视图中）
		listView.setAdapter(sp);
		//显示列表视图
		this.setContentView(listView);
	}
}
```

List 相当于 ListView 中的每一项，在使用 ArrayAdapter 做 ListView 适配器时，List 容器中只是简单添加了 String 字符串类型，但是这里需要在 ListView 的每一项中自定义组件，所以这里 List 声明不再是 List<String>，而是 List<Map<String, Object>>。

List<Map<String, Object>>可以理解为在 ListView 的每一项中不再是简单的一行字符串，而是将每一项添加一个组件容器 Map，在这个 Map 容器中放置自定义的组件。其 Map 的 put（String key, Object value）在进行添加数据时，每一个 put()函数都对应自定义 ListView 项中的一个组件；但是这里要注意按钮、复选框等组件是无法数据映射的。map.put（String key, Object value）的第一个参数用于初始化适配器时需要映射数据的对应索引；第二个参数表示对应自定义项布局中的组件数据。

代码中实例化 SimpleAdapter 适配器构造函数 SimpleAdapter（Context context, List data, int resource, String[] from, int [] to）的第一个参数是当前 context 对象，第二个参数是 ListView 各项数据，第三个参数是 ListView 每一项的布局，第四个参数是每一项中的数据映射索引数

组，第五个参数是每一项中数据对应的组件ID数组。

3.10.3　为ListView自定义适配器

前面说过按钮和复选框等这些附带事件的组件其实是无法将数据映射在 ListView 上的，所以如果需要监听和响应按钮、复选框等组件的事件时，则需要继承 BaseAdapter 进行自定义适配器来实现。下面就来对 ListView 自定义项布局中的按钮和复选框组件进行事件监听处理。

首先创建一个新类“MyAdapter.java”，使之继承 BaseAdapter 类，并且重写父类的 4 个抽象函数，代码如下：

```
public class MyAdapter extends BaseAdapter {
	public MyAdapter() {
	}
	@Override
	public int getCount() {
		return 0;
	}
	@Override
	public Object getItem(int position) {
		return null;
	}
	@Override
	public long getItemId(int position) {
		return 0;
	}
	@Override
	public View getView(int position, View convertView, ViewGroup parent)
	{
		return null;
	}
}
```

当一个 ListView 显示之前都会调用适配器中的 getCount()函数来确定 ListView 中项的长度，然后根据此长度再去调用 getView()函数绘制 ListView 中的每一项。

其实 ListView 中的适配器的作用就是将 ListView 每一项布局和组件进行实例化，并且设置组件的数据，所以主要去修改 getCount()与 getView()这两个方法即可，修改“MyAdapter.java”的代码如下：

```
public class MySimpleAdapter extends BaseAdapter {
	//声明一个LayoutInflater对象（其作用是用来实例化布局）
	private LayoutInflater mInflater;
	private List<Map<String, Object>> list;//声明List容器对象
```

```
	private int layoutID; //声明布局ID
	private String flag[];//声明ListView项中所有组件映射索引
	private int ItemIDs[];//声明ListView项中所有组件ID数组
	public MySimpleAdapter(Context context, List<Map<String,Object>>list,
			int layoutID, String flag[], int ItemIDs[]) {
		//利用构造来实例化成员变量对象
		this.mInflater = LayoutInflater.from(context);
		this.list = list;
		this.layoutID = layoutID;
		this.flag = flag;
		this.ItemIDs = ItemIDs;
	}
	@Override
	public int getCount() {
		return list.size();//返回ListView项的长度
	}

	@Override
	public Object getItem(int arg0) {
		return 0;
	}

	@Override
	public long getItemId(int arg0) {
		return 0;
	}
	//实例化布局与组件以及设置组件数据
	//getView(int position, View convertView, ViewGroup parent)
	//第一个参数：绘制的行数
	//第二个参数：绘制的视图这里指的是ListView中每一项的布局
	//第三个参数：view的合集，这里不需要
	@Override
	public View getView(int position, View convertView,ViewGroup parent){
		//将布局通过mInflater对象实例化为一个view
		convertView = mInflater.inflate(layoutID, null);
		for (int i = 0; i < flag.length; i++) {//遍历每一项的所有组件
			//每个组件都做匹配判断，得到组件的正确类型
			if (convertView.findViewById(ItemIDs[i]) instanceof
ImageView) {
				//findViewById()函数作用是实例化布局中的组件
				//当组件为ImageView类型，则为其实例化一个ImageView对象
				ImageView iv = (ImageView) convertView.
findViewById(ItemIDs[i]);
				//为其组件设置数据
				iv.setBackgroundResource((Integer) list.get
(position).get(
```

```
                                    flag[i]));
                    } else if (convertView.findViewById(ItemIDs[i])
instanceof TextView) {
                        //当组件为 TextView 类型，则为其实例化一个 TextView 对象
                        TextView tv = (TextView)
convertView.findViewById(ItemIDs[i]);
                        //为其组件设置数据
                        tv.setText((String)
list.get(position).get(flag[i]));
                    }
                }
                //为按钮设置监听

        ((Button)convertView.findViewById(R.id.btn)).setOnClickListener(
                        new View.OnClickListener() {
                            @Override
                            public void onClick(View v) {
                             //这里弹出一个对话框，后文有详细讲述
                             new AlertDialog.Builder(MainActivity.ma)
                            .setTitle("自定义 SimpleAdapter")
                            .setMessage("按钮成功触发监听事件！")
                            .show();
                            }
                        });
                //为复选框设置监听
                ((CheckBox)convertView.findViewById(R.id.cb)).
                setOnCheckedChangeListener(new OnCheckedChangeListener() {
                    @Override
                    public void onCheckedChanged(CompoundButton buttonView,
boolean isChecked) {
                        //这里弹出一个对话框，后文有详细讲述
                        new AlertDialog.Builder(MainActivity.ma)
                        .setTitle("自定义 SimpleAdapter")
                        .setMessage("CheckBox 成功触发状态改变监听事件！")
                        .show();
                    }
                });
                return convertView;
        }
}
```

关于对话框以及 findViewById()与 LayoutInflater 的区别，将在后文进行详细的讲解，这里就不再赘述。代码直接修改 MainActivity 类，将初始使用的 SimpleAdapter 修改成自定义的适配器 MySimpleAdapter，最后运行项目，效果如图 3-37 所示。



图 3-37　ListView 实现组件监听

这里要注意一点，因为 ListView 每一项中都有其他焦点的组件，例如定义的 Button 与 CheckBox，要解决这个问题很简单，只要将定义的 Button 与 CheckBox 焦点属性设置为不可见就可以了；修改布局文件中定义的 Button 与 CheckBox 组件，添加 focusable 属性，设置为 false：

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="button"
    android:id="@+id/btn"
    android:focusable="false"
    />
<CheckBox
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:id="@+id/cb"
   android:focusable="false"
   />
```

再次运行项目，效果如图 3-38 所示。



图 3-38　获得 ListView 中每一项的焦点

3.11 Dialog

在 Android 应用开发中，Dialog（对话框）创建简单且易于管理因而经常用到，对话框默认样式类似窗口样式的 Activitiy。

首先介绍 android.app.AlertDialog 下的 Builder 这个类。Builder 是 AlertDialog 类的子类，而且还是它的内部类。正如其名所示，Builder 相当于一个具体的构造者，通过 Builder 设置对话框属性，然后将 Builder（对话框）显示出来。

打开项目“DialogProject”，源代码为“3-11（Dialog 对话框）”，不需要修改布局，直接修改源代码 MainActivity.java：

```
public class MainActivity extends Activity {
        private Builder builder;//声明 Builder 对象
        @Override
        public void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);
                setContentView(R.layout.main);
                //实例化 Builder 对象
                builder = new Builder(MainActivity.this);
                //设置对话框的图标
                builder.setIcon(android.R.drawable.ic_dialog_info);
                //设置对话框的标题
                builder.setTitle("Dialog");
                //设置对话框的提示文本
                builder.setMessage("Dialog 对话框");
                //监听左侧按钮
                builder.setPositiveButton("Yes", new OnClickListener() {
                        @Override
                        public void onClick(DialogInterface dialog, int which) {
                        }
                });
                //监听右侧按钮
                builder.setNegativeButton("No", new OnClickListener() {
                        @Override
                        public void onClick(DialogInterface dialog, int which) {
                        }
                });
                //调用 show()方法显示出对话框
                builder.show();
        }
}
```

显示一个对话框，首先创建一个 Builder 对象，然后设置对话框属性，最后调用 show()方法将对话框显示出来。运行效果如图 3-39 所示。



图 3-39　对话框

对话框中最多可以设置显示 3 个按钮，左侧对应的是 PositiveButton，中间对应的是 NeutralButton，右侧对应的是 NegativeButton，在使用 Builder 对其按钮来设置监听器的时候也相当于设置了当前按钮显示可见，而且对话框默认只要单击了按钮此对话框就会消亡；Android 对 Builder 有特殊的设计模式，所以 OnCreate 函数的代码可以简写成以下形式：

```
new Builder(this).setIcon(android.R.drawable.ic_dialog_info).
setTitle("Dialog").
setMessage("Dialog 之\"二次确认\"样式").
setPositiveButton("Yes", new OnClickListener() {
      @Override
      public void onClick(DialogInterface dialog, int which) {
             // TODO Auto-generated method stub
      }
}).setNegativeButton("No", new OnClickListener() {
      @Override
      public void onClick(DialogInterface dialog, int which) {
             // TODO Auto-generated method stub
      }
}).create().show();
```

Builder 类中还有一个 setView()的方法，此方法是往对话框中添加系统组件；例如可以在对话框中添加一个 CheckBox 按钮：

```
builder.setView(new CheckBox(this));
```

运行效果如图 3-40 所示。

这里需要强调一点，在对话框中使用 setView()函数只能设置一个组件，如果多次使用 setView()也不会有错，但是先设置上的组件会被后设置的组件替换掉。

在对话框中除了可以设置一个组件之外，还可以同时添加单选框和复选框，其对应函数如下所示。



图 3-40　对话框中添加组件

（1）添加复选框的方法

M Builder.setMultiChoiceItems（String[]arg0,Boolean[]arg1,OnMultiChoiceClickListener arg3）

第一个参数：表示复选的各项文本；

第二个参数：表示复选的各项选中状态；

第三个参数：多选单击监听器。

（2）添加单选框方法

M Builder.setSingleChoiceItems（String[]arg0, int arg1,OnClickListener arg3）

第一个参数：表示单选的各项文本；

第二个参数：表示单选中默认选中的下标；

第三个参数：单击监听器。

首先在代码中添加一个单选按钮：

```
builder.setSingleChoiceItems(new String[]{"单选","单选"}, 1, new
OnClickListener() {
    @Override
    public void onClick(DialogInterface dialog, int which) {
        //which :选中项下标
    }
});
```

运行项目，效果如图 3-41 所示。



图 3-41　添加单选框（错误）

从效果图中看出，单选框并未正常显示；其实出现这种错误现象的原因是因为对话框的

提示文本和单选框布局发生冲突了，只须注释掉提示文本设置，然后再次运行项目，效果如图 3-42 所示。



图 3-42 添加单选框（正确）

图 3-42 正确显示了单选框，可以证实是因为设置了提示文本的缘故，那么复选框的布局也一样，也就是说对话框默认的布局中提示文本、单选框、复选框是相互冲突的。另外一点就是 setView()添加一个系统组件的布局，默认都放在对话框的最下方（即按钮上方）。

接下来，注释掉添加的单选按钮代码，添加如下复选框代码：

```
builder.setMultiChoiceItems(new String[] { "多选", "多选" },
 new boolean[] { false, true }, new OnMultiChoiceClickListener() {
      @Override
      public void onClick(DialogInterface dialog, int which, boolean
                          isChecked) {
                 //which：选中项下标
                 //isChecked：选中项的勾选状态
           }
      });
```

运行项目，效果如图 3-43 所示。



图 3-43 添加多选框

下面在对话框中添加一个简单的列表。注释掉复选框代码，添加如下的列表代码：

```
builder.setItems(new String[] { "列表项 1", "列表项 2", "列表项 3" }, new
```

```
OnClickListener() {
          @Override
          public void onClick(DialogInterface dialog, int which) {
               //which:选中项下标
          }
});
```

运行效果如图 3-44 所示。



图 3-44 添加列表

在对话框中，除了能添加这些系统组件以外，还可以添加自定义布局。其方法仍是使用 setView()函数，因为每个布局也是一个 View。为了让其布局在对话框中显示出来，首先新建一个布局文件“dialogmain.xml”，代码如下：

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
 android:layout_height="wrap_content" android:layout_width="wrap_content"
 android:background="#ffffffff" android:orientation="horizontal"
 android:id="@+id/myLayout">
    <TextView
      android:layout_height="wrap_content"
      android:layout_width="wrap_content"
      android:text="TextView" />
    <EditText
      android:layout_height="wrap_content"
      android:layout_width="wrap_content"
      />
    <Button
      android:layout_height="wrap_content"
      android:layout_width="wrap_content"
      android:text="btn1"
      />
    <Button
      android:layout_height="wrap_content"
```

```
        android:layout width "wrap content"
        android:text-" btn2"
        />
</LinearLayout>
```

线性布局为水平方向，其中注册了一个 TextView、一个 EditText 和两个 Button。

在源代码对话框设置布局之前，我们需要对新定义的整个布局进行实例化，否则在屏幕中什么都不会显示。对话框设置布局代码：

```
//实例 layout 布局
LayoutInflater inflater = getLayoutInflater();
View layout = inflater.inflate(R.layout.dialogmain,
                    (ViewGroup) findViewById(R.id.myLayout));
builder.setView(layout);
```

运行效果如图 3-45 所示。代码中 LayoutInflater 的作用是实例化一个布局，其详细用途在后文中有相应的讲解。



图 3-45　添加自定义布局

3.12 系统控件常见问题

3.12.1 Android 中常用的计量单位

Android 有时候需要一些计量单位，比如在布局 Layout 文件中可能需要指定具体单位等。常用的计量单位有：px、dip（dp）、sp，以及不常用的 pt、in、mm。下面详细介绍一下这些计量单位之间的区别与联系。

- in：英寸（长度单位）；
- mm：毫米（长度单位）；
- pt：磅/点，1/72 英寸（一个标准的长度单位）；
- sp：全名 scaled pixels—best for text size，放大像素，与刻度无关，可以根据用户的字

体大小进行缩放，主要用来处理字体的大小；

- px: 屏幕中的像素；
- dip（dp）：设备独立像素，一种基于屏幕密度的抽象单位；因为不同设备中有不同的显示效果，所以为了解决在不同分辨率手机上运行不至于相差太大的问题，引入了dip计量单位，这种计量单位与移动设备硬件无关。

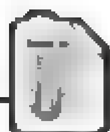
说到密度，这里简单介绍一下，手机密度值（Density）表示每英寸有多少个显示点，与手机的分辨率是两个概念，但是分辨率与密度之间又互相有关联，两者转换公式为：

- 密度值是120，屏幕实际分辨率为：240px × 400px（两个点对应一个分辨率）；
- 密度值是160，屏幕实际屏幕分辨率为：320px × 533px（3个点对应两个分辨率）；
- 密度值是240，屏幕实际屏幕分辨率为：480px × 800px（一个点对于一个分辨率）。

比如，QVGA与WQVGA屏的密度值是120，HVGA屏密度值是160，WVGA屏密度值是240。

在第2章介绍res资源目录的时候，曾经提到过，因为运行设备的不同，对应的资源文件目录也不同。其实真正的原因是，资源目录是根据密度的不同来进行划分的：

- 密度值是120，对应资源目录为drawable-ldpi;
- 密度值是160，对应资源目录为drawable-mdpi；
- 密度值是240，对应资源目录为drawable-hdpi。



提示

根据以上的介绍，在布局中应该尽量使用dip（dp）作为单位；而定义作为文字大小的单位则推荐使用sp。

3.12.2 Context

Context类是一个抽象类，它的子类很多，比如Activity、TabActivity、Service等。很多方法中需要传入Context参数才可实例对象，例如Toast实例对象时，第一个对象需传入Context对象。其实Context从字面上可理解为类似于句柄，联系上下文的意思。因为Activity是Context的子类，所以一般在Activity中使用Context的时候，可以用this来代替，但是如果在内部类中（如利用内部类使用监听组件的方式中），就不能使用this来代替Context，而是使用“ActName.this”，这里的ActName指的是Activity类的类名。

在Android中的Context可以有很多操作，但是最主要的功能是加载和访问资源；这里也只是对Context做一个简单的介绍，其更多的使用方式及说明可以参看Android提供的API文档。

3.12.3 Resources 与 getResources

在 Android 中资源（Resources）都会自动由 R.java 资源文件生成对应的静态 ID，通过 R 资源文件对资源生成的 ID 来引用。这样做的好处之前也做过说明，即在资源需要修改时，就不用去程序源代码中修改，直接修改对应 res 下的资源文件即可。

在源代码中，如果需要对资源目录下的 string.xml 中定义的字符串变量进行访问，只需要通过 getResources 的方式引用即可。

例如需要引用 string.xml 中的一个字符串，其变量名为“hello”，获取的方式如下：

```
getResources().getString(R.string.hello);
```

再如需要引用 drawable 目录下的一张名为“hello.png”的图片，获取的方式如下：

```
getResources().getDrawable(R.drawable.hello);
```

当然一些函数不仅支持传入 String 类型，也支持传入引用 ID。例如 TextView 中的 SetText()函数，这个方法不仅支持传入 String 类型，还支持 R 文件引用 ID 的参数。“R.string.strName”中的 strName 表示在 string.xml 中定义的字符串在 R 资源文件中生成的对应 ID 索引。

3.12.4 findViewById 与 LayoutInflater

LayoutInflater 的作用类似于 findViewById()，两者不同之处在于 LayoutInflater 是用来实例化 xml 布局文件中的布局；而 findViewById()，顾名思义，是通过 ID 来找到 xml 布局文件中定义的组件，比如 EditText、TextView、Button 等等。

关于用 LayoutInflater 来实例布局的方式有两种：

- 通过传入 Context 参数来获得 LayoutInflater 实例，然后调用 LayoutInflater 类中的 inflate 函数来得到布局实例。

```
LayoutInflater inflater = LayoutInflater.from(Context context);
View view=inflater.inflate(R.layout.ID, null);
```

- 通过系统服务来获取到 LayoutInflater 实例，然后调用 LayoutInflater 类中的 inflate 函数来得到布局实例。

```
LayoutInflater inflater = (LayoutInflater)getSystemService(LAYOUT_INFLATER
_SERVICE);
View view=inflater.inflate(R.layout.ID, null);
```

尽管实例布局的形式不同，但是这两种布局方式的性质没有区别。

3.12.5 多个 Activity 之间跳转/退出/传递数据操作

在介绍 Activity 生命周期的时候，介绍过多个 Activitiy 之间的跳转，但是没有详细讲解其实现方式。本小节将详细讲解一下 Activitiy 中常用的跳转、传递数据与退出操作。

首先新建一个项目“OpenOtherActivity”，源代码为“3-12-5（Activity 跳转与操作）”，修改布局代码如下：

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
      <TextView
          android:layout_width="fill_parent"
          android:layout_height="wrap_content"
          android:text="This is MainActivity!"
          />
      <Button
          android:layout_width="fill_parent"
          android:layout_height="wrap_content"
          android:text="OpenOtherActivity!"
          android:id="@+id/btnOpen"
          />
      <Button
          android:layout_width="fill_parent"
          android:layout_height="wrap_content"
          android:text="HideActivity"
           android:id="@+id/btnHideActivity"
          />
    <Button
          android:layout_width="fill_parent"
          android:layout_height="wrap_content"
          android:text="ExitActivity"
           android:id="@+id/btnExitActivity"
          />
</LinearLayout>
```

布局中注册一个 TextView 与三个 Button 组件，每个 Button 都定义了标题与 ID，然后再来修改 MainActivity 中的源代码：

```
public class MainActivity extends Activity implements OnClickListener {
      //声明按钮
      private Button btnOpen, btnHideActivity,btnExitActivity;
      @Override
```

```
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        //实例按钮
        btnOpen = (Button) findViewById(R.id.btnOpen);
        btnHideActivity = (Button) findViewById(R.id.btnHideActivity);
        btnExitActivity = (Button) findViewById(R.id.btnExitActivity);
        //给每个按钮添加监听
        btnOpen.setOnClickListener(this);
        btnHideActivity.setOnClickListener(this);
        btnExitActivity.setOnClickListener(this);
    }
    public void onClick(View v) {
        if (v == btnOpen) {
            //创建一个意图，并且设置需打开的Activity
            Intent intent = new Intent(MainActivity.this,
                                OtherActivity.class);
            //发送数据
            intent.putExtra("Main", "我是发送的数据~娃哈哈");
            //启动另外一个Activity
            this.startActivity(intent);
        } else if (v == btnHideActivity) {
            this.finish();//退出Activity
        }else if (v == btnExitActivity) {
            System.exit(0);//退出程序
        }
    }
}
```

上面的源代码实例化了三个按钮，并且在三个按钮上绑定了监听器，每个按钮在onClick()函数中都实现了处理事件代码。这里，对实例按钮与按钮绑定监听就不再赘述，只把注意力放在每个按钮的事件处理代码上。btnHideActivity 按钮事件调用 finish()函数，并退出当前的 Activity 操作；btnExitActivity 按钮事件使用 System.exit（0）函数，“退出程序”操作。

在代码中使用了 finish()和 System.exit（0）语句，它们的区别如下：

- finish()函数表示退出当前 Activity。执行此函数会调用生命周期中的 onStop()与 onDestory()函数，但是这仅仅是将当前的 Activity 推到后台，程序中的资源仍然存在，如果 Android 运行内存不是很紧张的情况下，程序是不会真正退出的；
- System.exit（0）函数表示退出当前的程序。当 Android 执行到此函数的时候，本应用程序的资源将被回收，并退出此程序。

这两种退出方式在手机进程中的对比如图 3-46 所示。



图 3-46　手机进程对比图

btnOpen 按钮事件用于打开另外一个 Activity 并且附带数据的传输。具体步骤如下：

步骤1 先声明了一个意图（Intent），实例意图的时候将需要打开的 Activity 类名设置在 Intent 中。

```
Intent intent = new Intent(Context packageContext, class cls);
```

第一个参数是 Context，第二个参数为被启动的 Activity 类。

```
intent.putExtra(String name, Object ob);
```

第一个参数是标识字符，类似与哈希表的 key 值，第二个参数根据需求填入一些基本类型，如 int、string、boolean 等等。

步骤2 然后使用当前 Context 的 startActivity（Intent intent）函数即可启动另外一个 Activity。

到此，MainActivity 的代码设计完毕，来看看新建的 Activity 类“OtherActivity. java”的源代码：

```
public class OtherActivity extends Activity {
      private TextView tv;
      @Override
      public void onCreate(Bundle savedInstanceState) {
             super.onCreate(savedInstanceState);
             tv = new TextView(this);
             setContentView(tv);
             //得到当前 Activity 的意图
             Intent intent = this.getIntent();
```

```
            //获取数据
            String str = intent.getStringExtra("Main");
            //将获取到的数据设置成 TextView 的文本
            tv.setText(str);
      }
}
```

在“OtherActivity”类中，为了获取传输过来的数据定义了意图对象。但是这里不要去new 一个新的意图，然后利用当前的 Activity 去得到意图的实例。获取数据的时候，要根据传输的数据类型，使用相对应的数据类型来获取。

打开另外一个 Activity，到此还缺少一个步骤，因为 OtherActivity 类继承的是 Activity，也是一个活动，那么之前讲过，当新建一个 Activity 的时候，需要在 AndroidManifest.xml 中声明，否则应用会由于找不到活动而报异常，修改 AndroidManifest.xml 如下：

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.openother"
      android:versionCode="1"
      android:versionName="1.0">
    <application android:icon="@drawable/icon"
android:label="@string/app_name">
        <activity android:name=".MainActivity"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category
                        android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <!-- 下面是注册新建的Activity(OtherActivity)-->
            <activity android:name=".OtherActivity"
                                  android:label="OtherActivity"/>
    </application>
</manifest>
```

AndroidManifest.xml 中声明了一个 Activity 活动。首先注意其语法层次，应该写在<application>标签的下一层，然后使用<activity>声明新的活动，最后设置新添加的 Activity 的类名与标题或者其他的属性。

单击“btnOpen”按钮事件，程序执行效果如图 3-47 所示。



图 3-47　打开另外一个 Activity 效果图

3.12.6　横竖屏切换处理的三种方式

Android 手机中运行应用的时候，一般用户都是竖屏，但是如果突然将手机横屏，那么很可能就会造成程序出现异常，因为在 Android 中每次屏幕切换会重启当前的 Activity。这种情况下，异常的解决方式有以下三种，对应的源代码为“3-12-6（横竖屏切换处理）”。

1. 锁定横竖屏切换

此方式只需要在 AndroidManifest.xml 文件中，对 Activity 定义屏幕方向属性只能为横屏或者竖屏即可。

- 将屏幕固定为竖屏显示：

```
<activity android:screenOrientation="portrait">
```

- 将屏幕固定为横屏显示：

```
<activity android:screenOrientation="landscape">
```

因为一个 Android 应用中可能会有多个 Activity，那么可以根据需要去配置每个 Activity 的显示方式，如果不设置，默认可以横竖屏切换。或者在源代码中设置横竖屏：

- 设置竖屏：

```
setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_PORTRAIT);
```

- 设置横屏：

```
setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_LANDSCAPE);
```

2．源代码中处理横竖屏切换事件

首先在 AndroidManifest.xml 中对 Activity 注册 android:configChanges 属性，然后在对应的 Activity 源代码中重写 onConfigurationChanged()函数即可。这样处理之后，当横竖屏切换的时候，就会响应其 Activity 中的 onConfigurationChanged()函数，然后对屏幕横竖屏做判定处理就可以了。

```
@Override
public void onConfigurationChanged(Configuration newConfig) {
    super.onConfigurationChanged(newConfig);
    if (this.getResources().getConfiguration().orientation ==
Configuration.ORIENTATION_LANDSCAPE) {
        Log.e("Himi", "当前屏幕切换成横屏显示模式");
    } else if (this.getResources().getConfiguration().orientation ==
Configuration.ORIENTATION_PORTRAIT) {
        Log.e("Himi", "当前屏幕切换成竖屏显示模式");
    }
}
```

使用此方式就不会在切换横竖屏的时候，Android 默认重启当前 Activity 了。

3. 重写 onSaveInstanceState()与 onRestoreInstanceState()函数

重写 onSaveInstanceState()与 onRestoreInstanceState()函数代码如下：

```
@Override
protected void onSaveInstanceState(Bundle outState) {
      super.onSaveInstanceState(outState);
      Log.e("Himi", "ONSAVE");
}
@Override
protected void onRestoreInstanceState(Bundle savedInstanceState) {
      super.onRestoreInstanceState(savedInstanceState);
      Log.e("Himi", "ONRESTORE");
}
```

在屏幕切换横竖屏的时候，会响应 onSaveInstanceState()函数，然后重启载入当前 Activity，最后响应 onRestoreInstanceState()函数，所以可以通过重写这两个函数，进行屏幕横竖屏切换时的处理。

以上 3 种处理横竖屏切换的方式，根据当前应用来进行选择处理。需要注意的是，使用 Android 模拟器测试，当横屏切换成竖屏时，第 2 种解决方式的 onConfigurationChanged()函数会响应两次。第 3 种解决方式会响应两遍 onSaveInstanceState()与 onRestoreInstance State()函数。不过大家放心，这只是 Android 模拟器的 bug，真机测试没有这种情况。

3.13 本章小结

本章主要介绍了布局与部分系统组件，重点在于对 Android 提供的 5 种布局要熟练掌握和灵活使用。本章介绍的组件都是在游戏中比较常用的，比如调整游戏的音量可以使用 SeekBar，帐号注册可以使用 TextVew 与 EditText 等等。对于 Android 中的其他组件的学习，请大家参考相关书籍。

第 4 章

游戏开发基础

从本章节可以学习到：

- ❖ 如何快速的进入 Android 游戏开发
- ❖ 游戏的简单概括
- ❖ Android 游戏开发中常用的三种视图
- ❖ View 游戏框架
- ❖ SurfaceView 游戏框架
- ❖ View 与 SurfaceView 的区别
- ❖ Canvas 画布
- ❖ Paint 画笔
- ❖ Bitmap 位图的渲染与操作
- ❖ 剪切区域
- ❖ 动画
- ❖ 游戏适屏的简述与作用
- ❖ 让游戏主角动起来
- ❖ 碰撞检测
- ❖ 游戏音乐与音效
- ❖ 游戏数据存储



4.1 如何快速的进入 Android 游戏开发

从本章开始我们正式进入 Android 游戏开发基础知识的学习。对于如何才能快速地学习 Android 游戏开发，本节跟大家分享一些编者的学习经验和方法。

1．不可盲目看 API 文档

很多人在接触学习一门新的平台语言时，总是喜欢先去探究一番 API 文档。先不说成效如何，至少编者认为这种方式不适合大部分人来效仿，主要原因在于 API 领域广泛，牵涉到的知识点太多，而对于刚刚接触平台开发语言的大部分人来说，遗忘的速度远远大于记忆！这种做法是大量消耗精力、小量吸取知识的方法，只会事倍功半。

2．前人铺路，后人乘凉

对于初学者来说，任何想要学习与掌握的知识点，之前都会有高人学习总结过；所以建议大家每学习一个知识点，都尽可能的先动手去网上搜索和学习别人总结出来的相关知识点的文章，毕竟前人总结过的知识会让你减少学习的弯路。最后再根据每个知识点去详细翻阅相关的 API 文档，有针对性、有目的性的去看 API 文档才会事半功倍。

3．好记性不如烂笔头

这句谚语，几乎无人不知无人不晓，但是总被许多初学者抛在脑后。在学习的时候，总是看代码的多，而动手练习代码的少！身为一个程序员都应该很清楚，代码如果不多动手敲它，它永远不会自己跑进脑中，所以多动手才是成功的关键。

4．养成自学的习惯

学习新的知识如果总是抱着依赖和期望别人手把手教授，那就太不现实了。因为没有任何一个人能时时刻刻的陪在身边给予帮助，但是使用 Baidu 和 Google 可以做到！它们拥有着最全的资源库，使用它们可以查找到最强的技术，不过，它们永远都只在那里等待你去使用它们，如果你不动手去搜索，那么对于你来说它们毫无用处！

5．利用小项目实战进步快

在学习游戏开发时，一定要多做小项目，比如今天学会了一个新的知识点，那么首先就要尽可能发散思维，多思考这个知识点会应用到什么类型的游戏中，并在游戏中起到什么样的作用等等。然后拿出时间一定要去写一个小项目练习新知识点。

写小项目有两点好处：一是巩固新知识点；二是通过小项目发现知识点实际应用到游戏中会出现的问题，有些问题不亲自动手编写是根本无法发现的。

6．进步来源于问题

好程序不是写出来的，是改出来的！这句话没有人能反驳，因为谁写代码都不可能是一遍成功，不用修改，不用完善的。

学习中遇到问题时，不应该烦躁而是应该庆幸，因为解决掉问题就意味着进步。千万不能遇到问题不假思索就去请教他人，这样解决掉的问题没有任何的意义，并且请教他人，就是在给他人创造一个学习或温习的机会！

当然这里不推荐大家遇到问题一定就铁下心的自己去几天几夜的钻研，应该自我把握问题的难易度，如果问题确实超出自己能力的，那请教他人反而对自己更有帮助，有效率，前提是自己考虑过如何解决此问题。

其实，游戏开发的学习过程应该是一个拼图的过程。首先要分模块来学习，积累了一定的模块知识后，再通过这些模块就可以拼出各种类型、各种风格的游戏。因此，后续章节中将以模块的形式来进行详解，并且最后会综合利用各个模块完成一个实战项目。下面我们就开始游戏开发的学习。

为了便于讲解，本章所有项目的创建统一选用 Android 1.6-API 4 版本；模拟器则统一使用如图 4-1 所示的属性配置，HVGA 默认索引资源文件夹为 drawable-mdpi。

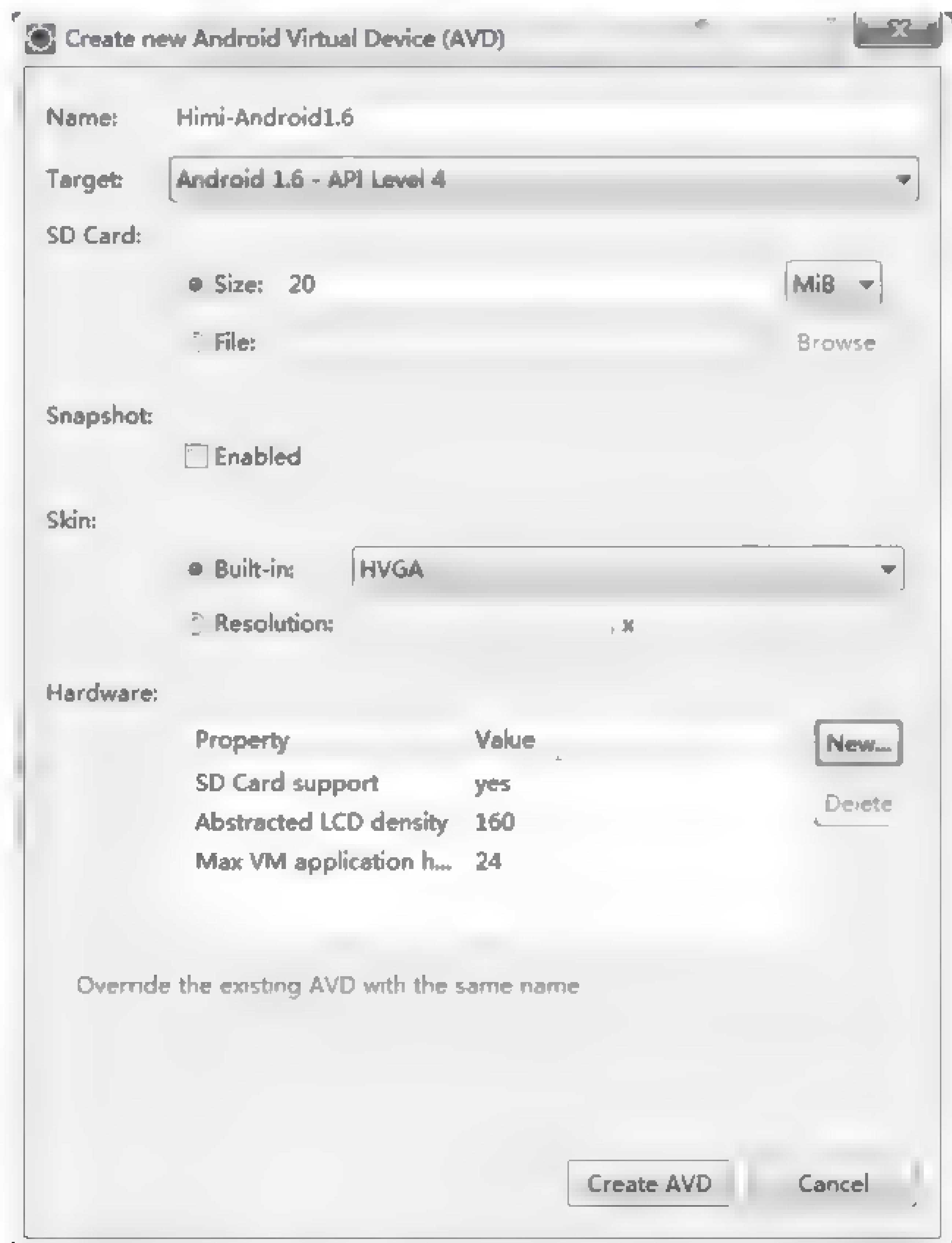


图 4-1　模拟器的属性配置

上一章为大家介绍了一些 Android 自带的常用组件，这些组件就像是 Google 为开发者提供的开发引擎一样，开发者只需要知道如何使用，并不要求知道其底层是如何实现的。从本章开始进入游戏开发，首先提醒读者注意的一点是：不管是否有过 Android 软件开发的经验，作者希望读者尽可能地忘却之前软件开发的流程和思维，换种眼光和思路去看待、去找到属于游戏开发专有的流程与设计思想。

在游戏开发中，一般很少使用系统提供的组件进行开发，其主要原因在于游戏的多样性。比如简单的一款“连连看”游戏，它就可以拥有 N 种玩法、N 种场景、N 种风格、N 种元素。所以，如果还期望从系统中找到对应组件的话，结果会令人很失望，不是系统不想提供，而是它永远都无法知道将要制作的游戏类型、风格等等。

总结一句话：开发一款游戏，请用自己的双手为这款游戏创建专属它的组件！换言之，就是要自己去实现游戏中的组件，不要再一味的幻想系统能为你带来什么。系统只能提供“一支笔”、“一张画布”，仅此而已。至于能创造出多么精彩的游戏世界，那完全取决于游戏开发者。

4.2 游戏的简单概括

对于玩家来说，游戏是动态的；对于游戏开发人员来说，游戏是静态的，只是在不停地播放不同的画面，让玩家看到了动态效果。

进入 Android 游戏开发之前，首先要熟悉三个重要的类：View（视图）、Canvas（画布）、Paint（画笔）。通过画笔，可以在画布上画出各种精彩的图形、图片等等，然后通过视图可以将画布上的内容展现在手机屏幕上。

其次要熟悉“刷屏”的概念。绘制在画布中的图像不管是图片还是图形，都是静态的，只有通过不断地展现不同的画布，才能实现动态的效果。在手机上，画布永远只是一张，所以不可能通过不断地播放不同的画布来实现动态效果，这时就需要对画布进行刷新来实现动态效果。

刷新画布如同使用一块橡皮擦，擦去之前画布上的所有内容，然后重新绘制画布，如此反复，形成动态效果，而擦拭画布的过程则称为刷屏（刷新屏幕）。刷屏的更多详细内容会在后文讲述，例如不刷屏带来的后果等等。下面我们就开始学习 Android 游戏开发中常用的三种视图吧。

4.3 Android 游戏开发中常用的三种视图

Android 游戏开发中常用三种视图是 View、SurfaceView 和 GLSurfaceView。这三种视图

的关系如图 4-2 所示。

```
java.lang.Object
  └android.view.View
      └android.view.SurfaceView
          └android.opengl.GLSurfaceView
```

图 4-2 三种视图关系树状图

下面简单介绍这三种视图的含义。

- View：显示视图，内置画布，提供图形绘制函数、触屏事件、按键事件函数等；
- SurfaceView：基于 View 视图进行拓展的视图类，更适用于 2D 游戏开发；
- GLSurfaceView：基于 SurfaceView 视图再次进行拓展的视图类，专用于 3D 游戏开发的视图。

另外使用 GLSurfaceView 支持 GPU 加速，通过真机测试发现，此视图在 GPU 加速下渲染 2D 图形方面的效率要远远高于 View 与 SurfaceView 30 倍左右；但 View 与 SurfaceView 的效率基本已经满足大部分游戏开发的要求，并且由于 GLSurfaceView 渲染位图都与 openGL 的知识相关，这里就不再多加讲解，有兴趣的可以参阅相关资料。

本书主要讲解的是 2D 游戏开发，所以 GLSurfaceView 类暂且不细说，而关于 View 与 SurfaceView 的区别，可以在大家熟悉这两种视图后再进行详细剖析，这里就简单说一句：View 与 SurfaceView 是 Android 游戏开发最常使用的两种视图。所以在 2D 游戏开发中，可以大致分为两种游戏框架，一种是 View 游戏框架，另外一种是 SurfaceView 游戏框架。

4.4 View 游戏框架

本节我们介绍 View 游戏框架。先来看一个范例，新建一个项目 GameView，如图 4-3 所示。本项目对应的源代码为“4-3（View 游戏框架）”。

项目创建完毕之后，首先自定义一个视图类“MyView”继承 View 类，代码如下：

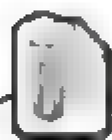


图 4-3　新建一个项目 GameView

```
public class MyView extends View {
      /**
       * 重写父类构造函数
       * @param context
       */
      public MyView(Context context) {
            super(context);
      }
      /**
       * 重写父类绘图函数
       */
      @Override
      protected void onDraw(Canvas canvas) {
            super.onDraw(canvas);
      }
      /**
       * 重写按键按下事件函数
       */
      @Override
```

```
    public boolean onKeyDown(int keyCode, KeyEvent event) {
        return super.onKeyDown(keyCode, event);
    }
    /**
     * 重写按键抬起事件函数
     */
    @Override
    public boolean onKeyUp(int keyCode, KeyEvent event) {
        return super.onKeyUp(keyCode, event);
    }
    /**
     * 重写触屏事件函数
     */
    @Override
    public boolean onTouchEvent(MotionEvent event) {
        return super.onTouchEvent(event);
    }
}
```

本类中实现的函数都是重写了父类 View 中的函数，当然 View 中的函数不止这些，可以根据需要来重写。比如在游戏开发中，最需要重写 View 的 onDraw 绘图函数，以及玩家对按键按下和抬起的事件监听函数 onKeyDown/onKeyUp、触屏事件监听函数 onTouchEvent 等。



提示

在 Eclipse 中重写父类函数，操作如下：单击主菜单的“Search”项，选中“Override/Implement Methods...”选项，然后在出现的“Override/Implement Method”窗口中，选中需要重写的函数，最后单击“OK”按钮即可（Eclipse 中默认使用 Shift+Alt+S 组合快捷键调出 Source 选项）。

下面修改 MainActivity 活动类让屏幕来显示 MyView 类：

```
public class MainActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        //设置显示 View 实例
        setContentView(new MyView(this));
    }
}
```

此时运行项目是看不到任何效果的，只能看到手机屏幕全黑，这是因为默认画布颜色为黑色。

4.4.1 绘图函数 onDraw

不管绘制文本、图形还是图片等等，首先需要的肯定是一个画布。而 View 类提供的实例中就有一个画布实例，这个画布实例存在于 View 的绘制函数 onDraw 的参数中，这也就是说，在画布上进行的一切绘制都应该添加到此绘制函数中。

我们尝试一下在画布上绘制文本。修改 MyView 类，在 onDraw 函数中添加如下代码：

```
protected void onDraw(Canvas canvas) {
    //创建一个画笔的实例
    Paint paint = new Paint();
    //设置画笔的颜色
    paint.setColor(Color.WHITE);
    //绘制文本
    canvas.drawText("Game", 10, 10, paint);
    super.onDraw(canvas);
}
```

在上面代码中，Paint 的 setColor 函数只有一个参数，传入的是一个 int 值。Color 是 Android 封装的颜色类，类中有很多静态颜色常量提供使用。当然此处的颜色值，也可以使用十六进制来表示的。"paint.setColor(Color.*WHITE*);" 等同于 "paint. setColor(0xffffffff);"。使用十六进制的好处是更灵活，因为使用十六进制来表示的 int 的四个字节分别表示透明度、红色分量、蓝色分量、绿色分量；这样不仅可以通过这个十六进制的数设置画笔透明度，也能设置画笔为各种需要的颜色。

Canvas 的 drawText 函数有四个参数。第一个参数：String 类型，指文本信息；第二个与第三个参数分别指文本绘制在屏幕中的 X、Y 位置坐标（默认绘制文字以文字的左下角为锚点）；第四个参数是画笔实例。

大家不要奇怪为什么绘制文本的函数不在 Paint 画笔类中，这是因为在 Android 中，绘制图形、图片等函数都是放在画布类里，由画布实例来调用这些函数进行绘图。而画布进行绘图时，画笔当然也是不可少的元素，只是这里画笔作为画布在绘图时以参数形式出现了。

运行项目，效果如图 4-4 所示。

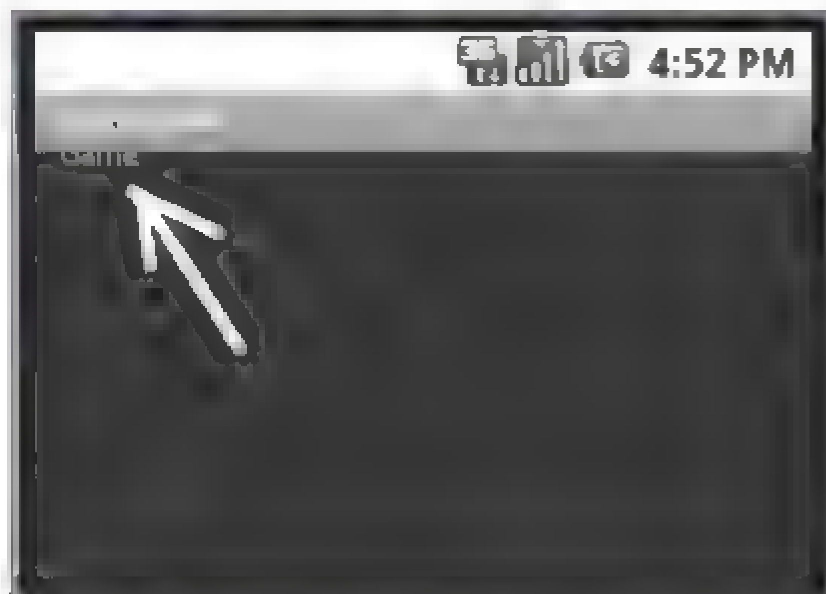


图 4-4 在屏幕中显示文本

图 4-4 箭头指向的“Game”字样就是绘制的文本了，从图中可以发现文本被应用名覆盖了，这是因为没有设置屏幕全屏的缘故。

下面为应用程序设置全屏。修改 MainActivity 类如下：

```
public class MainActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        //隐去标题栏（应用程序的名字）
        this.requestWindowFeature(Window.FEATURE_NO_TITLE);
        //隐去状态栏部分(电池等图标和一切修饰部分)
        this.getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULL
          SCREEN, WindowManager.LayoutParams.FLAG_FULLSCREEN);
        //设置显示 View 实例
        setContentView(new MyView(this));
    }
}
```

设置全屏的操作主要就两点：隐去状态栏部分，包括电池等图标；把应用的名字也隐去不显示。这样一来设置全屏就完成了，然后再次运行项目观察。要注意一点：设置隐去标题栏必须在显示 View 视图之前完成，否则程序会导致异常。

再次运行项目，效果如图 4-5 所示。



图 4-5 设置应用程序全屏

除了代码设置隐去应用标题和状态栏外，还可以通过在项目的 AndroidManifest.xml 文件中对 Activity 的属性进行配置来实现。

隐去应用标题：

```
android:theme="@android:style/Theme.NoTitleBar"
```

设置全屏（隐去状态栏和应用标题）：

```
android:theme="@android:style/Theme.NoTitleBar.Fullscreen"
```

这里再介绍一下手机屏幕XY坐标位置的小知识，手机屏幕XY坐标如图4-6所示。

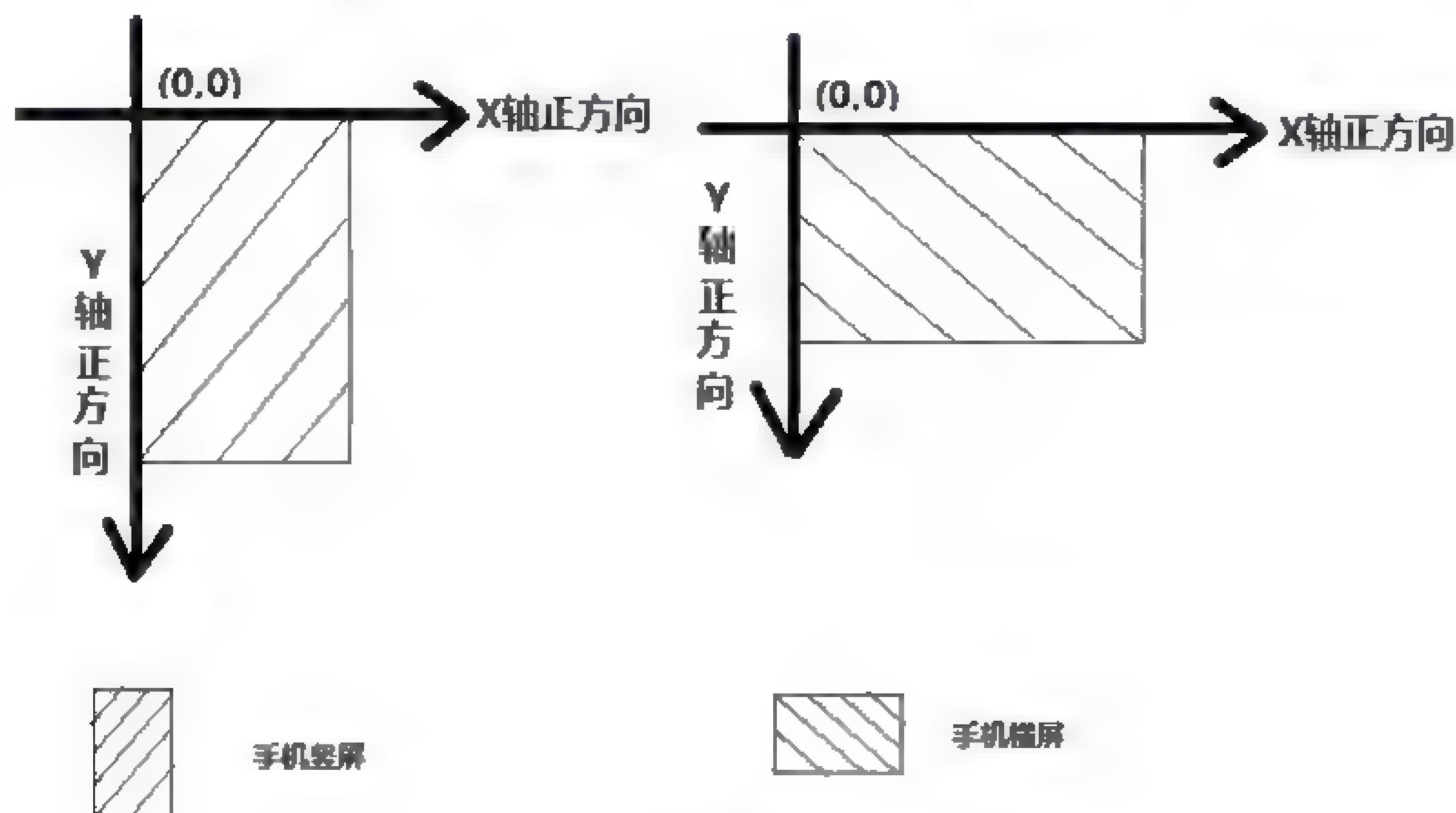


图4-6 手机横竖屏X、Y坐标轴

手机屏幕不论横屏，还是竖屏，手机的最左上角的点永远是（0,0）点；而手机屏幕的（0,0）点水平向右永远是X轴正方向，（0,0）点垂直向下永远是Y轴的正方向。

4.4.2 按键监听

在一款游戏中，与玩家交互的主要途径就是手机按键或玩家触摸屏幕这两种事件。在View视图类中已经封装了这些函数，只要重写按键、触屏监听函数即可获取当前玩家点击的是什么按键或者玩家点击屏幕的位置是哪里。

按键监听有两个函数：

- onKeyDown：按键被按下时响应的函数；
- onKeyUp：按键抬起时响应的函数。

触屏监听函数只有一个：

- onTouchEvent：触屏时响应的函数。

到这里大家可能会疑惑，实体按键监听分别对应有抬起和按下两种函数的监听，那么触屏事件为什么只有一个？手指按下屏幕和手指离开屏幕是如何监听的？其实触屏监听函数不只是玩家手指按下时触发响应，当手指离开屏幕、手指在屏幕中滑动等动作都可以通过此函数完成监听。

下面我们修改刚才的项目，实现让“Game”字样的文本通过实体方向按键控制其移动。

让文本的坐标随着方向按键的方向来移动，其实就是改变绘制文本的X、Y坐标；那么此时为了在按键监听函数中设置文本坐标，将文本的X、Y坐标定义为成员变量。首先定义文本的X、Y成员变量，并设置文本的初始坐标为（20,20）这个点：

```
private int textX=20,textY=20;
```

然后修改绘制函数如下：

```
@Override
protected void onDraw(Canvas canvas) {
        //创建一个画笔的实例
        Paint paint = new Paint();
        //设置画笔的颜色
        paint.setColor(Color.WHITE);
        //绘制文本
        canvas.drawText("Game", textX, textY, paint);
        super.onDraw(canvas);
}
```

这里要修改的就一点：将绘制文本的固定坐标换成了定义的成员坐标（textX, textY），这样一来只要改变 textX, textY 的值就可以了。最后编写按键事件监听函数的代码如下：

```
@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
      //判定用户按下的键值是否为方向键的"上下左右"键
      if (keyCode == KeyEvent.KEYCODE_DPAD_UP) {
             //"上"按键被点击，应该让文本的 Y 坐标变小
             textY-=2;
      } else if (keyCode == KeyEvent.KEYCODE_DPAD_DOWN) {
             //"下"按键被点击，应该让文本的 Y 坐标变大
             textY+=2;
       } else if (keyCode == KeyEvent.KEYCODE_DPAD_LEFT) {
             //"左"按键被点击，应该让文本的 X 坐标变小
             textX-=2;
       } else if (keyCode == KeyEvent.KEYCODE_DPAD_RIGHT) {
             //"右"按键被点击，应该让文本的 X 坐标变大
             textX+=2;
       }
       return super.onKeyDown(keyCode, event);
}
```

按键按下事件监听函数

```
public boolean onKeyDown(int keyCode, KeyEvent event) {}
```

其中，两个参数的含义如下：

- int keyCode：指的是当前用户点击的按键；
- KeyEvent event：指的是按键的动作事件队列，此类还定义了很多静态常量键值。

通过 keyCode 与手机键值的配对来确定当前用户的按键；一旦匹配到需要的键值后，就

可以处理响应的逻辑了。就像这里，如果希望玩家按下手机的四个方向按键后再去移动文本的位置，可以利用 keyCode（当前用户按下的按键键值）与所需的键值匹配来判定用户当前是否按下了对应的键值。一旦匹配键值相同，就开始分别处理四个方向按键应该处理的逻辑代码，这里是对文本的坐标进行修改。

到此为止就基本完成了按键控制文本移动的设计，但是遗憾的是，当运行项目后发现按下手机的四个方向键，文本没有任何的位置改变。文本位置没有发生改变这是正常的现象，因为当前的按键按下时，监听函数并没有在监听，原因在于没有设置当前 View 获取焦点。

所谓给当前 View 设置焦点，其实就是告诉系统，现在这个视图需要与用户交互，让系统来监听此视图。因为一个界面中可以显示多个 View 视图，Android 为了避免出现多个 View 焦点混乱的问题，View 类有了焦点属性，以及对应的设置焦点的方法。设置焦点的函数如下：

```
setFocusable(true);
```

此函数要求传入一个布尔值的参数，true 表示设置焦点，false 表示当前视图不需要焦点。视图只要设置一遍焦点即可。我们可以将设置焦点操作写在当前自定义的 MyView（View）构造函数中：

```
public MyView(Context context) {
    super(context);
    setFocusable(true);
}
```

此时再次运行项目，单击手机方向键，发现文本位置仍旧没有发生改变。别着急，这也是正常现象，下面解释其原因。

View 的 onDraw 函数虽然是绘制函数，但是此函数只会在 View 视图一开始创建运行的时候执行一遍而已。所以即使通过按键改变了绘制的文本坐标，但是看到的仍然是之前的画布，不是最新的画布状态，如果想看到最新的画布则需要重新绘制画布。

重新绘制画布的方法，View 类也提供了相应的两个函数，而且这两个函数都会再次调用 onDraw 函数。

- invalidate()
- postInvalidate()

这两个重新绘制画布的函数的主要区别是：invalidate()方法不能在当前线程中循环调用执行，这里所说的线程不是系统的主 UI 线程，而是指的子线程（自己创建的线程）；而 postInvalidate()函数可以在子线程中循环调用执行。如果不在当前 View 创建线程循环重绘画布的话，这两种重绘画布的函数就没什么区别了，都可以使用。

知道了重绘画布的方法之后，紧接着要考虑的是，重绘函数方法添加在哪里，这个问题其实要根据实际情况而定了。比如当前要做的是一个让文本移动的效果，那么改变文本的坐

标只是在按键后被修改，所以可以将重绘函数加到按键监听的函数中。这种情况下，如果要求游戏的画布每隔固定时间刷新一次，那么可能就需要另起一个线程不断地循环调用重绘画布了。

在按键按下函数中添加重绘函数：

```
@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
      …
      invalidate();
      //postInvalidate();
      return super.onKeyDown(keyCode, event);
}
```

这两种重绘函数，开发人员可以按自己的喜好选择。

到此为止，让一个文本通过方向按键移动的效果就真正的完成了。这个过程貌似很复杂，其实是很简单的，为了让大家更加深刻地理解，讲解的内容稍微会偏多一点；希望大家学习到此，自己动手练习一下，熟悉相关的过程和细节。

运行项目，效果如图 4-7 所示。



图 4-7　按键监听

按键监听除了 onKeyDown 还有 onKeyUp，两种监听区别在于事件的状态，一个是按下，一个是抬起，其按键键值的匹配方法都是一样的；那么到底使用哪种按键监听，就需要根据实际的项目来选择了。

这里需要提醒的一点是，当前在做的是让绘制的文本移动的效果，其实它是个动态的效果，在本章开始的时候就解释过，动态效果有两种方式实现：

- 不断的绘制新的画布；
- 使用一张画布，通过刷屏来让这张画布恢复到初始空白画布的状态，然后再向画布上进行绘制。

本小节通过调用 View 类提供的重绘函数来实现文本的动态效果，那么 View 封装的这两

种重绘函数，底层实现到底是重新在新的画布上绘制，还是利用刷屏来实现的，有兴趣的朋友可以下载 Android 的 SDK 源码研究一下。这里需要记住的是，不管 Android 是通过哪种方式封装的重绘函数，只要知道使用了重绘函数就不要再去做刷屏操作即可。至于如何用代码实现刷屏，将会在后续讲解 SurfaceView 游戏框架时详细为大家介绍。

4.4.3 触屏监听

上一小节利用按键监听实现了文本的移动，本小节将利用触屏监听函数来实现让文本跟随玩家手指在屏幕的位置移动。仔细分析一下，让文本跟随手指移动，无疑有两种情况：

- 手指点击屏幕时，文本的 X、Y 坐标要在手指相对于屏幕的位置；
- 手指在屏幕上滑动，文本的 X、Y 坐标跟随手指在屏幕的位置移动。

其实不管哪种情况，总结一句来说就是：文本的坐标永远是玩家手指在手机屏幕上的位置！

为了实现文本跟随手指移动的效果，我们在触屏监听函数中添加代码如下：

```
@Override
public boolean onTouchEvent(MotionEvent event) {
	int x = (int)event.getX();
	int y = (int)event.getY();
	//玩家手指点击屏幕的动作
	if (event.getAction() == MotionEvent.ACTION_DOWN) {
		textX = x;
		textY = y;
		//玩家手指抬起离开屏幕的动作
	} else if (event.getAction() == MotionEvent.ACTION_MOVE) {
		textX = x;
		textY = y;
		//玩家手指在屏幕上移动的动作
	} else if (event.getAction() == MotionEvent.ACTION_UP) {
		textX = x;
		textY = y;
	}
	//重绘画布
	invalidate();
	//postInvalidate();
	return super.onTouchEvent(event);
}
```

在上面代码中使用了触屏事件监听函数：

```
public boolean onTouchEvent(MotionEvent event) {}
```

触屏监听函数只有一个参数 MotionEvent event。此类实例中保存了玩家触屏的动作，比如常见的动作有：按下动作、抬起动作、移动动作、屏幕压力、多点触屏等等，当然此类中也定义了很多动作的静态常量值。通过 event.getAction()方法获取玩家的动作与所需动作常量值匹配。

运行项目，使用手指点击屏幕、离开屏幕都很正常，但是当手指在屏幕中进行滑动的时候，文本坐标并没有跟随手指移动，也就是说 MotionEvent.*ACTION_MOVE* 的动作，系统获取不到。这里简单解释一下原因：onTouchEvent()函数通常情况下会去执行 super.onTouchEvent()函数并传回布尔值。但是 super.onTouchEvent()中的 super 会有可能并没做任何事，并且回传 false 回来。一旦回传 false 回来，后面的 event 动作可能就会收不到了，所以为了确保后面的 event 能顺利收到，应该让触屏监听函数的返回值永远为 true。因此，修改触屏监听函数如下：

```
@Override
public boolean onTouchEvent(MotionEvent event) {
        …
        return true;
}
```

针对当前做的文本跟随用户手指的功能，在触屏监听函数中，其实没有必要获取用户的动作，因为不管用户是什么动作，开发人员需要的只是用户手指触摸在屏幕上的 X、Y 坐标位置，所以修改以将触屏监听函数简化：

```
@Override
public boolean onTouchEvent(MotionEvent event) {
        //获取用户手指触屏的 X 坐标赋值与文本的 X 坐标
        textX = (int)event.getX();
        //获取用户手指触屏的 Y 坐标赋值与文本的 Y 坐标
        textY = (int)event.getY();
        //重绘画布
        invalidate();
        //postInvalidate();
        return true;
}
```

其实 View 中触屏监听函数也对应有一个设置触屏焦点的函数：

```
setFocusableInTouchMode(true);
```

但是默认不用设置，触屏监听函数也能正常响应。

到此为止，对 View 视图的按键监听、触屏监听、绘制函数、重绘函数、画布的绘制等都做了相应的解释和说明。其实在游戏开发中需要系统支持的无非就是这些函数了，所以在 Android 的 View 视图中进行游戏开发基本就讲解完了，至此，整个 View 的游戏框架也就介

绍完毕了。至于如何在画布中绘制图形、图片，以及设置画笔属性等内容将在讲述完SurfaceView游戏框架后再进行详细的讲解。

4.5 SurfaceView 游戏框架

上一节讲解了View游戏框架，本节将继续讲解SurfaceView游戏框架。

4.5.1 SurfaceView 游戏框架实例

新建项目“GameSurfaceView”，如图4-8所示。本项目对应的源代码为“4-4（SurfaceView游戏框架）”。

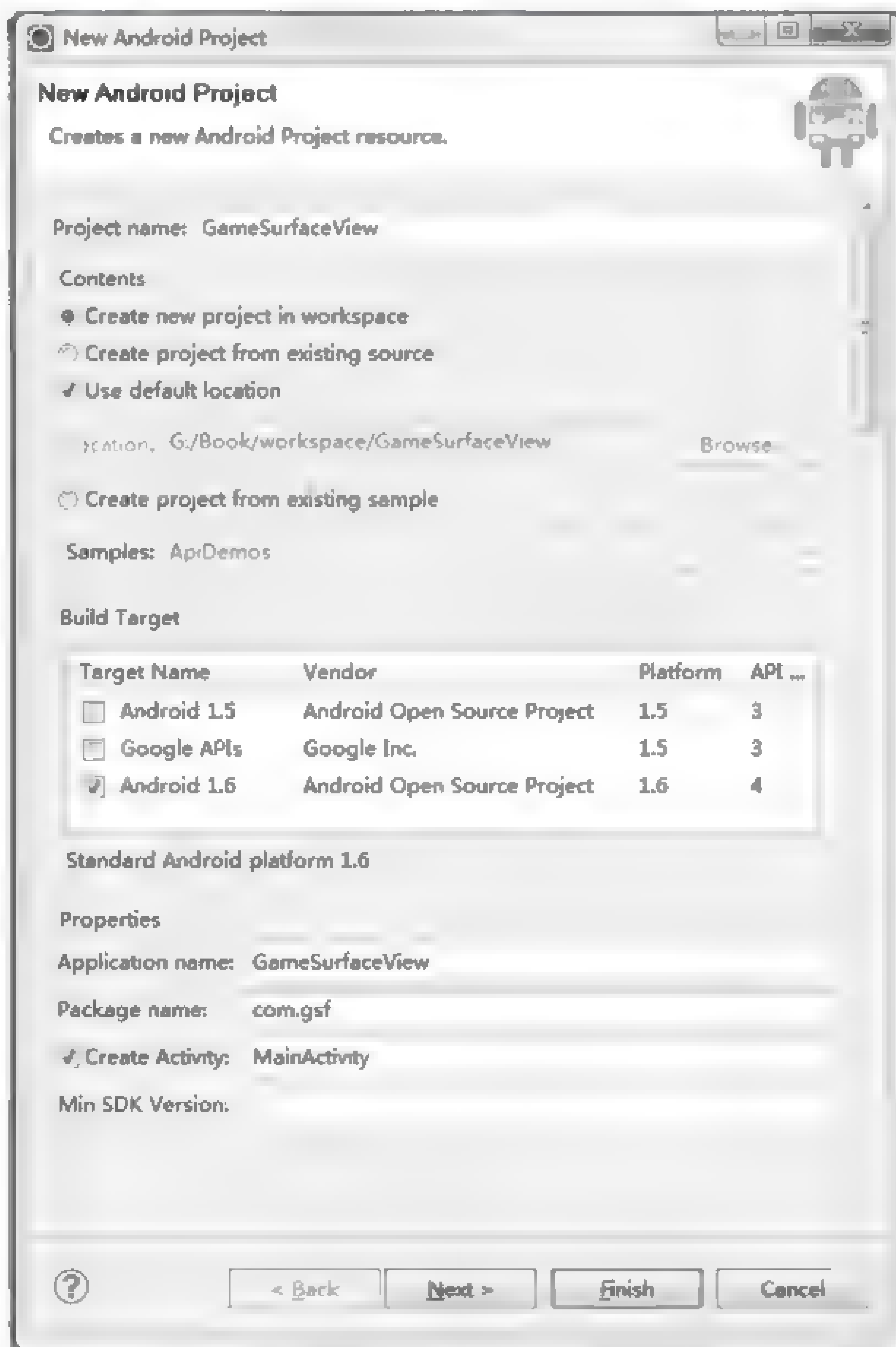


图4-8 新建项目“GameSurfaceView”

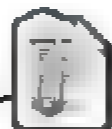
首先自定义一个类“MySurfaceView”，此类继承 SurfaceView，除此之外还要实现 android.view.SurfaceHolder.Callback 接口，代码如下：

```
public class MySurfaceView extends SurfaceView implements Callback {
	//用于控制 SurfaceView
	private SurfaceHolder sfh;
	private Paint paint;
	public MySurfaceView(Context context) {
		super(context);
		//实例 SurfaceHolder
		sfh = this.getHolder();
		//为 SurfaceView 添加状态监听
		sfh.addCallback(this);
		//实例一个画笔
		paint = new Paint();
		//设置画笔颜色为白色
		paint.setColor(Color.WHITE);
	}
	@Override
	public void surfaceCreated(SurfaceHolder holder) {
		myDraw();
	}
	@Override
	public void surfaceChanged(SurfaceHolder holder, int format, int
width, int height) {
	}
	@Override
	public void surfaceDestroyed(SurfaceHolder holder) {
	}
	/**
	 * 自定义绘图函数
	 */
	public void myDraw() {
		Canvas canvas = sfh.lockCanvas();
		canvas.drawText("Game", 10, 10, paint);
		sfh.unlockCanvasAndPost(canvas);
	}
}
```

在上面程序中定义了一个 SurfaceHolder 类的实例，此类提供控制 SurfaceView 的大小、格式等，并且主要用于监听 SurfaceView 的状态。其实 SurfaceView 只是保存当前视图的像素数据，在使用 SurfaceView 时，并不会与 SurfaceView 直接打交道，而是通过 SurfaceHolder 来控制，使用 SurfaceHolder 的 lockCanvas()函数来获取到 SurfaceView 的 Canvas 对象，再通过在 Canvas 上绘制内容来修改 SurfaceView 中的数据。

lockCanvas()函数不仅是获取 Canvas，同时还对获取的 Canvas 画布进行加锁，这里对画

布进行同步加锁的机制主要是为了防止 SurfaceView 在绘制过程中被修改、摧毁等发生的状态改变；与 lockCanvas()函数对应的还有一个 unlockCanvasAndPost（Canvas canvas）函数用于解锁画布和提交。



注意

SurfaceHolder 类除了 lockCanvas()函数可以获取当前视图的画布（画布默认大小同手机屏幕大小）外，还提供一个 lockCanvas（Rect rect）函数，其中传入一个 Rect 矩形类的实例，用于得到一个自定义大小的画布。

SurfaceHolder 对 SurfaceView 的状态进行监听需要使用 android.view.SurfaceHolder.Callback 接口；此接口需要重写三个函数，用于监听 SurfaceView 的不同状态：

（1）当 SurfaceView 被创建完成后响应的函数

```
@Override
public void surfaceCreated(SurfaceHolder holder) {
}
```

（2）当 SurfaceView 状态发生改变时响应的函数

```
@Override
public void surfaceChanged(SurfaceHolder holder, int format, int
      width, int height) {
}
```

（3）当 SurfaceView 状态摧毁时响应的函数

```
@Override
public void surfaceDestroyed(SurfaceHolder holder) {
}
```

最后通过 SurfaceHolder 类的 addCallback（CallBack callback）函数将其监听接口实例传入，可完成对 SurfaceView 的状态监听。

SurfaceView 是 View 的子类，所以也拥有按键监听函数、触屏监听函数等这些父类方法；但是值的注意的是因为 SurfaceView 是通过 SurfaceHolder 来修改其数据，所以在 SurfaceView 上进行绘制不再使用 onDraw（Canvas canvas）来绘图，而是通过 SurfaceHolder 获取到 SurfaceView 的 Canvas，然后再进行绘制；所以即使重写 View 的 onDraw（Canvas canvas）函数，在 SurfaceView 启动时也不会执行到。

因此，这里自定义了一个绘图函数：

```
public void myDraw() {
      Canvas canvas = sfh.lockCanvas();
      canvas.drawText("Game", 10, 10, paint);
```

```
    sfh.unlockCanvasAndPost(canvas);
}
```

此方法通过 SurfaceHolder 的 lockCanvas()函数得到一个 Canvas 实例，然后绘制文本，最后解锁并提交画布。接下来，修改 MainActivity 类，让其显示自定义的 SurfaceView 视图：

```
public class MainActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        //设置全屏
    this.getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCR
EEN, WindowManager.LayoutParams.FLAG_FULLSCREEN);
        requestWindowFeature(Window.FEATURE_NO_TITLE);
        //显示自定义的 SurfaceView 视图
        setContentView(new MySurfaceView(this));
    }
}
```

最后运行项目，效果如图 4-9 所示。



图 4-9　SurfaceView 视图

运行项目后发现 SurfaceView 视图正常显示，那么下面来实现在讲解 View 游戏框架时的一个小功能“让文本‘Game’跟随玩家触屏的手指移动”。

首先定义文本的坐标为成员变量：

```
private int textX=10,textY=10;
```

然后修改绘制函数：

```
public void myDraw() {
    Canvas canvas = sfh.lockCanvas();
    canvas.drawText("Game", textX, textY, paint);
    sfh.unlockCanvasAndPost(canvas);
```

```
}
```

最后完成触屏监听（重写 View 的触屏监听函数）：

```
@Override
public boolean onTouchEvent(MotionEvent event) {
      textX = (int) event.getX();
      textY = (int) event.getY();
      myDraw();
      return true;
}
```

触屏事件中的代码，用于获取当前玩家触屏的坐标赋值与文本的坐标，最后调用绘图函数 myDraw()函数来重新绘图。

运行项目，效果如图 4-10 所示。



图 4-10　画布截图

通过图 4-10 所示的效果图来看，用户手指在屏幕中滑动后视图显示简直就是一团糟。在介绍 View 游戏框架完成这个“文本跟随玩家手指移动”的小功能时，对屏幕的重绘都是使用了 View 提供的重绘函数进行的，因为使用 View 的两种重绘函数后，会默认重新调用 View 的 onDraw 函数。那么，既然在触屏事件中也调用了自定义的绘图函数，为什么这里就出现问题了呢？

仔细观察图 4-10 也不难看出，画布肯定更新到了最新的状态，否则文本的最新位置不会显示，但是没有显示正常的效果，视图中应该是只存在一个“Game”的文本字样，其原因是

画布没有刷新，将每次绘制的文本全部都显示了出来。解决方法就是对画布进行“刷屏”。

4.5.2 刷屏的方式

在之前使用 View 视图时并没有手动“刷屏”，其原因也解释过，是因为 View 类本身提供的两种重绘函数，其内部已经封装了对画布的刷屏操作（也可能每次都绘制在一个新的画布上），所以每次在 onDraw（Canvas canvas）中重绘画布永远看不到之前绘制过的图形。

但是，在当前使用的 SurfaceView 是自定义的绘图函数，而且每次获取到的 Canvas 仍然是上次使用过的画布。系统没有刷新画布，也没有重新提供一张画布，我们在每次绘制之前也没进行刷屏，这样相当于我们不断地在一张画布上进行绘图，必然会遗留下以前画布的状态。

所以，使用 SurfaceView 视图时，在得到其画布 Canvas 之后，首先进行的应该是刷屏操作，将画布上绘制的图形全部清空，然后再进行绘图。

刷屏的方式有以下几种：

（1）每次绘图之前，绘制一个等同于屏幕大小的图形覆盖在画布上。

修改绘图函数如下：

```
public void myDraw() {
      Canvas canvas = sfh.lockCanvas();
      //绘制矩形(画笔默认为填充)
      canvas.drawRect(0,0,this.getWidth(),this.getHeight(), paint);
      canvas.drawText("Game", textX, textY, paint);
      sfh.unlockCanvasAndPost(canvas);
}
```

设置画笔为填充样式，画笔默认绘制图形不填充；

这样每次在画布上绘图前都绘制一个填充的，大小等同于屏幕的矩形覆盖画布，只要这个矩形的颜色等同于屏幕默认的颜色，那就等同于将屏幕做了清空操作。

（2）每次绘图之前，在画布上填充一种颜色。

修改绘图函数如下：

```
public void myDraw() {
      Canvas canvas = sfh.lockCanvas();
      canvas.drawColor(Color.BLACK);
      canvas.drawText("Game", textX, textY, paint);
      sfh.unlockCanvasAndPost(canvas);
}
```

Canvas 类中的 drawColor(int color)函数是往整个画布中填充一种颜色。

（3）每次绘图之前，指定 RGB 来填充画布。

修改绘图函数如下：

```
public void myDraw() {
        Canvas canvas = sfh.lockCanvas();
        canvas.drawRGB(0, 0, 0);
        canvas.drawText("Game", textX, textY, paint);
        sfh.unlockCanvasAndPost(canvas);
}
```

Canvas 类中的 drawRGB（int r,int g,int b）函数是指定一种颜色对屏幕进行填充，其方法的三个参数分别是红色、绿色、蓝色组合而成的颜色的分量，当三个值都为 0 时，就是黑色；三个值都是 255 时，就是白色。

（4）每次绘图之前，绘制一张等同于屏幕大小的图片覆盖在画布上。

很多游戏都会有背景图，所以每次在画布上绘制，首先绘制背景图即可，但是这张背景图一定要等同于屏幕的大小，否则屏幕部分区域肯定还会看到以前绘图的遗迹。

其实以上的四种刷屏方式虽然使用的方法不一，但是原理都一样，就是每次在画布绘制之前都对画布进行一次整体的覆盖。

4.5.3 SurfaceView 视图添加线程

在游戏中，基本上不会等用户每次触发了按键事件、触屏事件才去重绘画布，而是会固定一个时间去刷新画布；比如游戏中的倒计时、动态的花草、流水等等，这些游戏元素并不会跟玩家交互，但是这些元素却都是动态的。所以游戏开发中，都会有一个线程不停的去重绘画布，实时的更新游戏元素的状态。

当然游戏中除了画布给玩家最直接的动态展现外，也会有很多逻辑需要不间断地去更新，比如怪物的 AI（人工智能）、游戏中钱币的更新等等。

下面就为本节实例项目中的 SurfaceView 视图添加线程，用于不停的重绘画布以及不停地执行游戏逻辑。完整的 SurfaceView 游戏框架中的自定义视图 MySurfaceView 类代码如下：

```
public class MySurfaceView extends SurfaceView implements Callback,
Runnable {
        //用于控制SurfaceView
        private SurfaceHolder sfh;
        //声明一个画笔
        private Paint paint;
        //文本的坐标
        private int textX = 10, textY = 10;
        //声明一条线程
        private Thread th;
        //线程消亡的标识位
        private boolean flag;
        //声明一个画布
```

```
private Canvas canvas;
//声明屏幕的宽高
private int screenW, screenH;
/**
 * SurfaceView 初始化函数
 */
public MySurfaceView(Context context) {
        super(context);
        //实例 SurfaceHolder
        sfh = this.getHolder();
        //为 SurfaceView 添加状态监听
        sfh.addCallback(this);
        //实例一个画笔
        paint = new Paint();
        //设置画笔颜色为白色
        paint.setColor(Color.WHITE);
        //设置焦点
        setFocusable(true);
}
/**
 * SurfaceView 视图创建，响应此函数
 */
@Override
public void surfaceCreated(SurfaceHolder holder) {
        screenW = this.getWidth();
        screenH = this.getHeight();
        flag = true;
        //实例线程
        th = new Thread(this);
        //启动线程
        th.start();
}
/**
 * 游戏绘图
 */
public void myDraw() {
        try {
                canvas = sfh.lockCanvas();
                if (canvas != null) {
                        //-----------利用绘制矩形的方式，刷屏
                        ////绘制矩形
                        //canvas.drawRect(0,0,this.getWidth(),
                        //this.getHeight(), paint);
                        //-----------利用填充画布，刷屏
                        //              canvas.drawColor(Color.BLACK);
                        //-----------利用填充画布指定的颜色分量，刷屏
```

```
                    canvas.drawRGB(0, 0, 0);
                    canvas.drawText("Game", textX, textY, paint);
                }
            } catch (Exception e) {
                // TODO: handle exception
            } finally {
                if (canvas != null)
                    sfh.unlockCanvasAndPost(canvas);
            }
    }
    /**
     * 触屏事件监听
     */
    @Override
    public boolean onTouchEvent(MotionEvent event) {
            textX = (int) event.getX();
            textY = (int) event.getY();
            return true;
    }
    /**
     * 按键事件监听
     */
    @Override
    public boolean onKeyDown(int keyCode, KeyEvent event) {
            return super.onKeyDown(keyCode, event);
    }
    /**
     * 游戏逻辑
     */
    private void logic() {
    }
    @Override
    public void run() {
            while (flag) {
                long start = System.currentTimeMillis();
                myDraw();
                logic();
                long end = System.currentTimeMillis();
                try {
                    if (end - start < 50) {
                        Thread.sleep(50 - (end - start));
                    }
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
```

```
	}
	/**
	 * SurfaceView 视图状态发生改变，响应此函数
	 */
	@Override
	public void surfaceChanged(SurfaceHolder holder, int format, int
width, int height) {
	}
	/**
	 * SurfaceView 视图消亡时，响应此函数
	 */
	@Override
	public void surfaceDestroyed(SurfaceHolder holder) {
		flag = false;
	}
}
```

本类中有很多需要注意的地方，按照代码由上到下的顺序详解说明：

（1）线程标识位

在代码中“boolean flag;”语句声明一个布尔值，它主要用于以下两点：

①便于消亡线程

大家都知道一个线程一旦启动，就会执行其 run()函数，run()函数执行结束后，线程也伴随着消亡。由于游戏开发中使用的线程一般都会在 run()函数中使用一个 while 死循环，在这个循环中会调用绘图和逻辑函数，使得不断的刷新画布和更新逻辑；那么如果游戏暂停或者游戏结束时，为了便于销毁线程在此设置一个标识位来控制。

②防止重复创建线程及程序异常

为什么会重复创建线程，首先从 Android 系统的手机说起。熟悉或者接触过 Android 系统的人都知道，Android 手机上一般都会有“Back（返回）”与“Home（小房子）”按键；不管当前手机运行了什么程序，只要单击“Back”或者“Home”按键的时候，默认会将当前的程序切入到系统后台运行（程序中没有截获这两个按钮的前提下）；也正因为如此，会造成 SurfaceView 视图的状态发生改变。下面来讲解这两个按钮按下以及重新回到程序时，SurfaceView 都执行到了哪些函数。

首先单击“Back”按钮使当前程序切入后台，然后单击项目重新回到程序中，SurfaceView 的状态变化为：surfaceDestroyed→构造函数→surfaceCreated →surfaceChanged。

然后单击“Home”按钮使当前程序切入后台，单击项目重新回到程序中，SurfaceView 的状态变化为：surfaceDestroyed→surfaceCreated→surfaceChanged。

通过 SurfaceView 的状态变化可以明显看到，当点击“Back”按键并重新进入程序的过程要比点击“Home”按键多执行了一个构造函数。也就是说，当点击“Back”返回按键时，SurfaceView 视图会被重新加载。

正因为这个原因，如果线程的初始化是在视图构造函数或者在视图构造函数之前，那么

线程启动也要放在视图构造函数中进行。

千万不要把线程的初始化放在 surfaceCreated 视图创建函数之前，而线程的启动却放在 surfaceCreated 视图创建的函数中，否则程序一旦被玩家点击“Home”按键后再重新回到游戏时，程序会抛出异常，异常信息如下：

```
java.lang.IllegalThreadStateException: Thread already started
    at java.lang.Thread.start(Thread.java:1286)
    at com.gsf.MySurfaceView.surfaceCreated(MySurfaceView.java:62)
    at android.view.SurfaceView.updateWindow(SurfaceView.java:392)
    at android.view.SurfaceView.onWindowVisibilityChanged(SurfaceView.java:182)
    at android.view.View.dispatchWindowVisibilityChanged(View.java:3745)
    at android.view.ViewGroup.dispatchWindowVisibilityChanged(ViewGroup.java:690)
    at android.view.ViewGroup.dispatchWindowVisibilityChanged(ViewGroup.java:690)
    at android.view.ViewRoot.performTraversals(ViewRoot.java:694)
    at android.view.ViewRoot.handleMessage(ViewRoot.java:1613)
    at android.os.Handler.dispatchMessage(Handler.java:99)
    at android.os.Looper.loop(Looper.java:123)
    at android.app.ActivityThread.main(ActivityThread.java:4203)
    at java.lang.reflect.Method.invokeNative(Native Method)
    at java.lang.reflect.Method.invoke(Method.java:521)
    at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:791)
    at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:549)
    at dalvik.system.NativeStart.main(Native Method)
```

异常是因为线程已经启动造成的，原因很简单，因为程序被“Home”键切入后台再从后台恢复时，会直接进入 surfaceCreated 视图创建函数中，又执行了一遍线程启动！

能够想到的解决方法是，可以将线程的初始化和启动都放在视图的构造函数中，或者都放在视图创建的函数中。但是这里又出现新的问题，如果将线程的初始化和启动都放在视图的构造函数中，那么当程序被“Back”键切入后台再从后台恢复时，线程的数量会增多，反复多次，就会反复多出对应的线程。

那么，大家可能又会想到将 flag 这个线程标识位在视图摧毁时让其值改为 false，从而使当前这个线程的 run 方执行完毕，以达到摧毁掉线程的目的。不幸的是，这也是错误的做法。

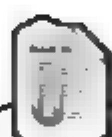
大家可以想想，即使在视图销毁时利用 flag 标识位摧毁游戏线程，但是如果点击“Home”按键呢？当程序恢复的时候，程序就不执行线程了，也就是说重绘和逻辑函数都不再执行！

所以最完美的做法就是，线程的初始化与线程的启动都写在视图的 surfaceCreated 创建函数中，并且将线程标识位在视图摧毁时将其值改变为 false。这样既可以避免“线程已启动”的异常，还可以避免点击 Back 按键无限增加线程数的问题。

（2）获取视图的宽和高

在 SurfaceView 视图中获取视图的宽和高的方法：

- this.getWidth()：获取视图宽度。
- this.getHeight()：获取视图高度。



注意

> 在 SurfaceView 视图中获取视图的宽高，一定要在视图创建之后才可获取到，也就是在 surfaceCreated 函数之后获取，在此函数执行之前获取到的永远是零，因为当前视图还没有创建，是没有宽高值的。

（3）绘图函数 try 一下

因为当 SurfaceView 不可编辑或尚未创建时，调用 lockCanvas()函数会返回 null；Canvas 进行绘图时也会出现不可预知的问题，所以要对绘制函数中进行 try...catch 处理；既然 lockCanvas()函数有可能获取为 null，那么为了避免其他使用 canvas 实例进行绘制的函数报错，在使用 Canvas 开始绘制时，需要对其进行判定是否为 null。

（4）提交画布必须放在 finally 中

绘图的时候可能会出现不可预知的 Bug，虽然使用 try 语句包起来了，不会导致程序崩溃；但是一旦在提交画布之前出错，那么解锁提交画布函数则无法被执行到，这样会导致下次通过 lockCanvas()来获取 Canvas 时程序抛出异常，原因是因为画布上次没有解锁提交！所以画布将解锁提交的函数应放入 finally 语句块中。

还要注意，虽然这样保证了每次能正常提交解锁画布，但是提交解锁之前要保证画布不为空的前提，所以还需判断 Canvas 是否为空，这样一来就完美了。

（5）刷帧时间尽可能保证一致

虽然在线程循环中，设置了休眠时间，但是这样并不完善！比如在当前项目中，run 的 while 循环中除了调用绘图函数还一直调用处理游戏逻辑的 logic()函数，虽然在当前项目的逻辑函数中并没有写任何的代码，但是假设这个逻辑函数 logic()中写了几千行的逻辑，那么系统在处理逻辑时，时间的开销是否与上次的相同，这是无法预料的，但是可以尽可能地让其时间差值趋于相同。假设游戏线程的休眠时间为 X 毫秒，一般线程的休眠写法为：

```
Thread.sleep(X);
```

优化写法步骤如下：

步骤1 首先通过系统函数获取到一个时间戳：

```
long start = System.currentTimeMillis();
//在线程中的绘图、逻辑等的函数
```

步骤2 处理以上所有函数之后，再次通过系统函数获取到一个时间戳：

```
long end = System.currentTimeMillis();
```

步骤3 通过这两个时间戳的差值，就可以知道这些函数所消耗的时间：如果（end - start）>X，那线程就完全没有必要去休眠；如果（end - start）<X，那线程的休眠时间应

该为 X-（end – start）。

线程休眠应更改为以下写法：

```
if((end - start)<X){
      Thread.sleep(X-(end - start));
}
```

一般游戏中刷新时间在 50~100 毫秒之间，也就是每秒 10~20 帧左右；当然还要视具体情况和项目而定。

4.6 View 与 SurfaceView 的区别

在 Android 2D 游戏开发中，可以选用 View 与 SrufaceView 这两种视图进行开发，前面简单的讲解了 View 和 SurfaceView 游戏框架后，本节就来总结一下两者的区别和特点，从而让大家在开发游戏前根据游戏的类型来选择合适的视图。

1．更新画布

在 View 视图中对于画布的重新绘制，是通过调用 View 提供的 postInvalidate()与invalidate()这两个函数来执行的，也就是说画布是由系统主 UI 进行更新。那么当系统主 UI 线程更新画布时可能会引发一些问题；比如更新画面的时间一旦过长，就会造成主 UI 线程被绘制函数阻塞，这样一来则会引发无法响应按键、触屏等消息的问题。

SurfaceView 视图中对于画布的重绘是由一个新的单独线程去执行处理，所以不会出现因主 UI 线程阻塞而导致无法响应按键、触屏信息等问题。

2．视图机制

Android 中的 View 视图是没有双缓冲机制的，而 SurfaceView 视图却有！也可以简单理解为，SurfaceView 视图就是一个由 View 扩展出来的更加适合游戏开发的视图类。

简单介绍了这两点区别，貌似都在说 SurfaceView 视图的好，其实不然，因为 View 与 SurfaceView 都各有其优点；比如一款棋牌类的游戏，此类型游戏画面的更新属于被动更新；因为画布的重绘主要是依赖于按键与触屏事件（当玩家有了操作之后画布才需要进行更新），所以此类游戏选择 View 视图进行开发比较合适，而且也减少了因使用 SurfaceView 需单独起一个新的线程来不断更新画布所带来的运行开销。

但如果是主动更新画布的游戏类型，比如 RPG、飞行射击等类型的游戏中，很多元素都是动态的，需要不断重绘元素状态，这时再使用 View 显然就不合适了。所以到底开发游戏使用哪种视图更加的合适，这完全取决于游戏类型、风格与需求。

总体来说，SurfaceView 更加适合游戏开发，因为它能适应更多游戏类型；在后文讲解的

项目中都将使用 SurfaceView 游戏框架进行开发。

为了便于讲解，这里首先声明两点：

① 只要后文中提到“使用 SurfaceView 游戏框架”则表示项目中包含两个类：

- MainActivity.java：用于设置全屏，显示游戏视图
- MySurfaceView.java：自定义游戏视图类，继承 SurfaceView；并且 MySurfaceView 类中包含游戏开发常用的按键、触屏、绘图、逻辑等函数。

② 本章中运行项目的 Android 模拟器在没有特别声明的情况下，一律采用如图 4-1 所示配置参数的模拟器运行项目。

4.7 Canvas 画布

画布类 Canvas 封装了图形与图片绘制等内容，此类常用的函数说明如下：

M drawColor（int color）

作用：绘制颜色覆盖画布，常用于刷屏

参数：颜色值，也可用十六进制形式表示（ARGB）

M drawText（String text, float x, float y, Paint paint）

作用：绘制文本字符

第一个参数：文本内容

第二、三个参数：文本的 X，Y 坐标

第四个参数：画笔实例

M drawPoint（float x, float y, Paint paint）

作用：绘制像素点

第一、二个参数：像素的 X,Y 坐标

第三个参数：画笔实例

M drawPoints（float[] pts, Paint paint）

作用：绘制多个像素点

第一个参数：Float 数组，数组中放置的是多个像素点的 X,Y 坐标

第二个参数：画笔实例

M drawLine（float startX, float startY, float stopX, float stopY, Paint paint）

作用：绘制一条直线

前两个参数：起始点的 X,Y 坐标

后两个参数：终点的 X,Y 坐标

最后一个参数：画笔实例

M drawLines（float[] pts, Paint paint）

作用：绘制多条直线

第一个参数：Float 数组，数组中放置的是多个直线的起始点与终点 X，Y 坐标

第二个参数：画笔实例

- drawRect（float left, float top, float right, float bottom, Paint paint）

  作用：绘制矩形

  第一、二个参数：矩形的左上角 X,Y 坐标

  第三、四个参数：矩形的右下角 X,Y 坐标

  第五个参数：画笔实例

- drawRect（Rect r, Paint paint)

  作用：绘制矩形

  第一个参数：矩形实例

  第二个参数：画笔实例

- drawRoundRect（RectF rect, float rx, float ry, Paint paint)

  作用：绘制圆角矩形

  第一个参数：矩形实例

  第二个参数：圆角 X 轴的半径

  第三个参数：圆角 Y 轴的半径

  第四个参数：画笔实例

- drawCircle（float cx, float cy, float radius, Paint paint）

  作用：绘制圆形

  第一、二个参数：圆形的中心点 X,Y 坐标

  第三个参数：圆形的半径

  第四个参数：画笔实例

- drawArc（RectF oval, float startAngle, float sweepAngle, boolean useCenter, Paint paint)

  作用：绘制弧形（扇形）

  第一个参数：矩形实例

  第二个参数：弧形的起始角度（默认 45°为图形的起始角度 0°）

  第三个参数：弧形的终止角度

  第四个参数：是否绘制中心点：如果为真，起始点与终止点都会分别连接中心点，从而形成封闭图形；如果为假，则起始点直接连接终止点，从而形成封闭图形；

  第五个参数：画笔实例

- drawOval（RectF oval, Paint paint)

  作用：绘制椭圆

  第一个参数：矩形实例

  第二个参数：画笔实例

- drawPath（Path path, Paint paint)

作用：绘制指定路径图形

第一个参数：路径实例

第二个参数：画笔实例

M drawTextOnPath（String text, Path path, float hOffset, float vOffset, Paint paint)

作用：将文本沿着指定路径进行绘制

第一个参数：文本

第二个参数：路径实例

第三个参数：文本距离绘制起点的距离

第四个参数：文本距离路径的距离

第五个参数：画笔实例

这些函数都比较容易理解，它们所使用的参数中需要特别介绍的是 Rect 与 Path 这两个类：

- Rect：矩形类，利用两个点的坐标从而确定矩形的大小；

其常用的构造函数为：

M Rect（float left, float top, float right, float bottom）

第一、二个参数表示矩形的左上角坐标；

第三、四个参数表示矩形的右下角坐标。

Android 中还提供了一个 RectF 类，RectF 类与 Rect 类主要的区别是长度单位精确度不同；RectF 使用单精度浮点数，而 Rect 使用 int 类型；在使用 Canvas 绘制矩形时，可以直接传入矩形的四个参数的方式，也可以选择传入一个矩形实例。

- Path：指定绘制的路径，然后按照其路径的路线依次绘制，组合任意需要的图形。

其常用函数如下：

M moveTo（float x, float y）

作用：设定路径的起始点

两个参数：起始点的坐标

M lineTo（float x, float y）

作用：以上次的终点作为起点，以本次的坐标点为终点，两点之间使用一条直线连接

两个参数：本次点线的终点位置

M close()

作用：路径结束的标识，如果路径关闭前的点不是起点，将自动连接封闭

以上的 moveTo、lineTo 与 close 三个函数搭配使用，路径起点与终点只需要设置一次，而路线 lineTo 则可以设置多个。

M android.graphics.Path.quadTo（float x1, float y1, float x2, float y2）

作用：绘制贝赛尔曲线

第一个参数：操作点的 x 坐标

第二个参数：操作点的 y 坐标

第三个参数：结束点的 x 坐标

第四个参数：结束点的 y 坐标

以上这个函数是 Android 实现并封装好的贝赛尔曲线方法，为了让大家知道如何使用，对应的也编写了一个在视图中绘制贝赛尔曲线的项目，这里不进行详细讲解了，对应的源代码为“4-7-1（贝塞尔曲线）”。

- addArc（RectF oval, float startAngle, float sweepAngle）
- addOval（RectF oval, Direction dir)
- addCircle（float x, float y, float radius, Direction dir）
- addRect（RectF rect, Direction dir)
- addRoundRect（RectF rect, float[] radii, Direction dir)

以上函数都是在添加不同图形的绘制路径，比如添加圆形路径、矩形路径、椭圆路径等等；虽然这种直接添加图形路径的方法相对于使用 moveTo、lineTo 与 close 这种组合路径方法大大减少了工作量，但是直接添加图形路径的方法并没有组合路径灵活，至于想绘制什么形状的图形，还要根据具体情况来做择优选择。

到此为止，对于 Canvas 一些最常用的函数，都已经做了解释和说明，下面就通过实例代码实现来观察这些常用函数的显示效果。

首先新建项目“CanvasProject”，游戏框架为 SurfaceView 游戏框架，项目对应的源代码为“4-7-2（Canvas 画布）”。修改 MySurfaceView 类的绘图函数代码如下：

```
public void myDraw() {
       try {
              canvas = sfh.lockCanvas();
              if (canvas != null) {
                     //----利用填充画布，刷屏
                     canvas.drawColor(Color.BLACK);
                     //----绘制文本
                     canvas.drawText("drawText", 10, 10, paint);
                     //----绘制像素点
                     canvas.drawPoint(10, 20, paint);
                     //----绘制多个像素点
                     canvas.drawPoints(new float[] { 10, 30, 30,30}, paint);
                     //----绘制直线
                     canvas.drawLine(10, 40, 50, 40, paint);
                     //----绘制多条直线
                     canvas.drawLines(new float[] { 10, 50, 50, 50, 70, 50,
                                   110, 50 }, paint);
                     //----绘制矩形
                     canvas.drawRect(10, 60, 40, 100, paint);
```

```
                //----绘制矩形 2
                Rect rect = new Rect(10, 110, 60, 130);
                canvas.drawRect(rect, paint);
                canvas.drawRect(rect, paint);
                //----绘制圆角矩形
                RectF rectF = new RectF(10, 140, 60, 170);
                canvas.drawRoundRect(rectF, 20, 20, paint);
                //----绘制圆形
                canvas.drawCircle(20, 200, 20, paint);
                //----绘制弧形
                canvas.drawArc(new RectF(150, 20, 200, 70), 0, 230,
                        true, paint);
                //----绘制椭圆
                canvas.drawOval(new RectF(150, 80, 180, 100), paint);
                //----绘制指定路径图形
                Path path = new Path();
                //设置路径起点
                path.moveTo(160, 150);
                //路线 1
                path.lineTo(200, 150);
                //路线 2
                path.lineTo(180, 200);
                //路径结束
                path.close();
                canvas.drawPath(path, paint);
                //----绘制指定路径图形
                Path pathCircle = new Path();
                //添加一个圆形的路径
                pathCircle.addCircle(130, 260, 20, Path.Direction.CCW);
                //----绘制带圆形的路径文本
                canvas.drawTextOnPath("PathText", pathCircle, 10, 20,
                            paint);
            }
        } catch (Exception e) {
            // TODO: handle exception
        } finally {
            if (canvas != null)
                sfh.unlockCanvasAndPost(canvas);
        }
}
```

项目运行效果如图 4-11 所示。

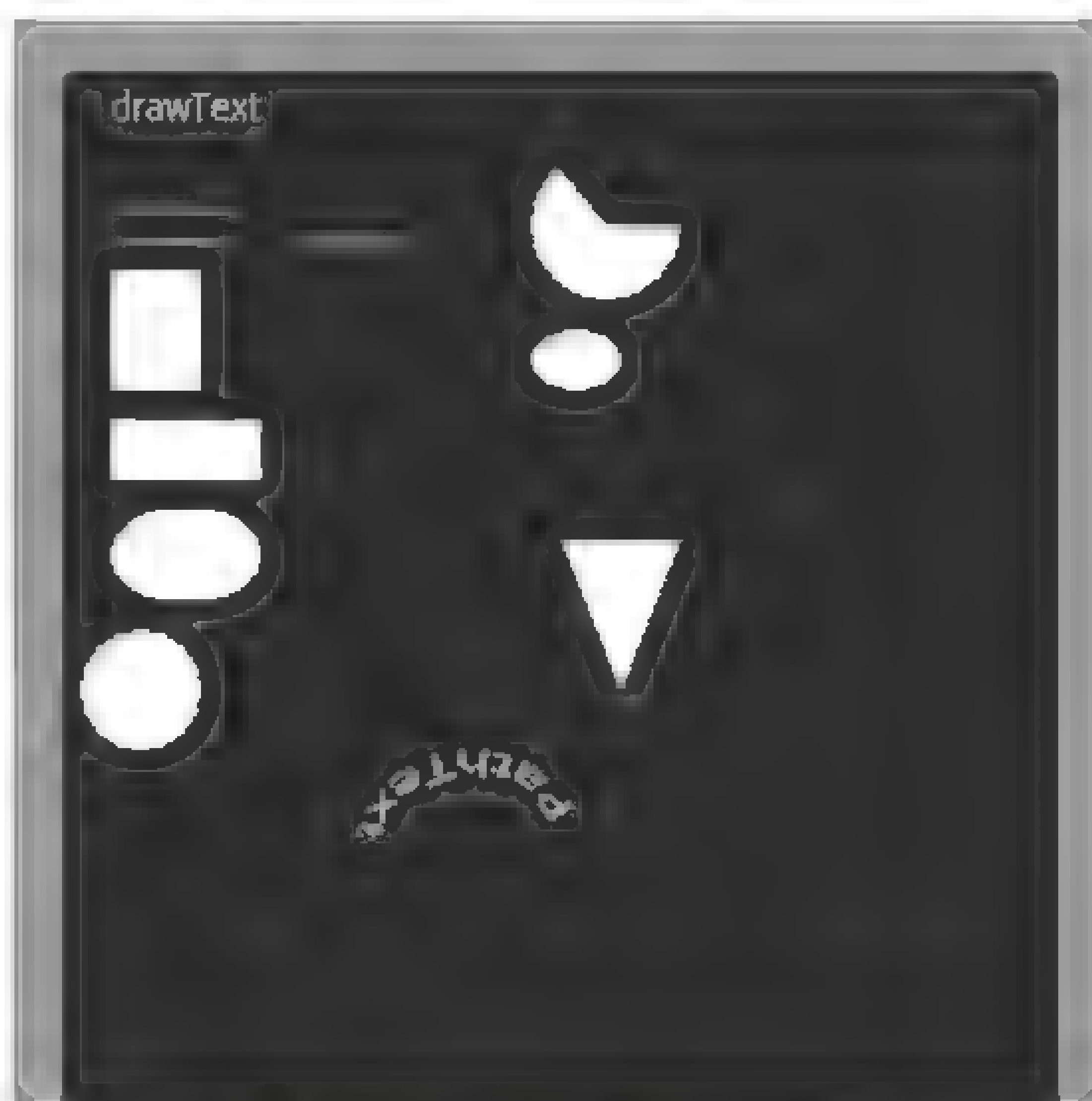


图 4-11　Canvas 常用函数练习

4.8 Paint 画笔

Paint（画笔）是绘图的辅助类，其类中包含文字与位图的样式、颜色等属性信息。Paint 的常用方法如下：

- setAntiAlias（boolean aa）

  作用：设置画笔是否无锯齿

  参数：true 表示无锯齿，false 表示有锯齿，默认为 false。

  可以通过观察如图 4-12 所示的效果图，更加形象地解释此方法的作用。

- setAlpha（int a）

  作用：设置画笔透明度

  参数：透明值

- setTextAlign（Paint.Align align）

  作用：设置绘制文本的锚点

  参数：Paint.Align类中的常量



图 4-12 画笔无锯齿

- measureText（String text）

  作用：获取文本内容的宽度

  参数：文本内容

- setStyle（Style style）

  作用：设置画笔样式

  参数：样式实例

- setColor（int color）

  作用：设置画笔颜色

  参数：色值

- setStrokeWidth（float width）

  作用：设置画笔的粗细程度

  参数：画笔粗细值

- setTextSize（float textSize）

  作用：设置画笔在绘制文本时，文本字体的尺寸

  参数：尺寸值

- setARGB（int a, int r, int g, int b）

  作用：设置画笔的 ARGB 分量

  第一个参数：画笔透明度分量

  第二个参数：画笔红色分量

  第三个参数：画笔绿色分量

  第四个参数：画笔蓝色分量

熟悉了 Paint 的常用函数后，通过实例代码深入理解这个类。新建项目“PaintProject”，游戏框架为 SurfaceView 游戏框架，项目对应的源代码为“4-8（Paint 画笔）”。修改 MySurfaceView 类的绘图函数如下：

```
public void myDraw() {
    try {
```

```
        canvas = sfh.lockCanvas();
        if (canvas != null) {
            canvas.drawColor(Color.WHITE);
            //-----设置画笔无锯齿
            Paint paint1 = new Paint();
            canvas.drawCircle(40, 30, 20, paint1);
            paint1.setAntiAlias(true);
            canvas.drawCircle(100, 30, 20, paint1);
            //-----设置画笔的透明度
            canvas.drawText("无透明度", 100, 70, new Paint());
            Paint paint2 = new Paint();
            paint2.setAlpha(0x77);
            canvas.drawText("半透明度", 20, 70, paint2);
            //-----设置绘制文本的锚点
            canvas.drawText("锚点", 20, 90, new Paint());
            Paint paint3 = new Paint();
            //设置以文本的中心点绘制
            paint3.setTextAlign(Paint.Align.CENTER);
            canvas.drawText("锚点", 20, 105, paint3);
            //------获取文本的长度
            Paint paint4 = new Paint();
            float len =paint4.measureText("文本宽度:");
            canvas.drawText("文本长度:"+len, 20, 130, new Paint());
            //------设置画笔样式
            canvas.drawRect(new Rect(20,140,40,160), new Paint());
            Paint paint5 = new Paint();
            //设置画笔不填充
            paint5.setStyle(Style.STROKE);
            canvas.drawRect(new Rect(60,140,80,160), paint5);
            //-------设置画笔颜色
            Paint paint6 = new Paint();
            paint6.setColor(Color.GRAY);
            canvas.drawText("灰色", 30, 180, paint6);
            //------设置画笔的粗细程度
            canvas.drawLine(20, 200,70, 200, new Paint());
            Paint paint7 = new Paint();
            paint7.setStrokeWidth(7);
            canvas.drawLine(20, 220,70, 220,paint7);
            //------设置画笔绘制文本的字体粗细
            Paint paint8 = new Paint();
            paint8.setTextSize(20);
            canvas.drawText("文字尺寸", 20, 260, paint8);
            //------设置画笔的 ARGB 分量
            Paint paint9 = new Paint();
            paint9.setARGB(0x77, 0xff, 0x00, 0x00);
            canvas.drawText("红色半透明", 20, 290, paint9);
```

```
            }
        } catch (Exception e) {
            // TODO: handle exception
        } finally {
            if (canvas != null)
                sfh.unlockCanvasAndPost(canvas);
        }
}
```

项目运行效果如图 4-13 所示。



图 4-13　画笔练习

Paint 画笔类提供了一个抗锯齿的函数，其实 Canvas 画布也提供了绘图抗锯齿的函数，如下所示：

M Canvas.setDrawFilter（DrawFilter filter）；

作用：为画布设置绘图抗锯齿

参数：绘图过滤器实例

实例化一个DrawFilter类的对象，代码如下所示：

```
PaintFlagsDrawFilter pfd = new PaintFlagsDrawFilter(0,
Paint.ANTI_ALIAS_FLAG | Paint.FILTER_BITMAP_FLAG);
```

4.9 Bitmap 位图的渲染与操作

Bitmap 是图形类，Android 系统支持的图片格式有 png、jpg、bmp 等。

对位图操作在游戏中是很重要的知识点，比如游戏中需要两张除了大小之外其他完全相同的图，那么如果会对位图进行缩放操作，很容易就节约了一张图片资源；这样既节约了美工的时间，更节约了游戏安装包的大小；当然除了缩放之外，还有很多操作，例如对位图进

行旋转、镜像、设置透明等等操作都会节约很大的资源。

首先创建一张位图实例。位图的实例不能通过 new，如果想通过一张图片资源文件创建一个位图，则要通过位图工厂来索引图片资源文件，从而生成一张位图实例，如下所示：

M BitmapFactory .decodeResource（Resources res,int Id）

作用：通过资源文件生成一张位图

第一个参数：资源实例

第二个参数：资源 ID

懂得如何通过图片资源创建位图实例后，下面就来详细介绍如何操作位图。创建项目“BitmapProject”，游戏框架为 SurfaceView 游戏框架，对应的源代码为“4-9（Bitmap 位图渲染与操作）”。

修改 MySurfaceView.java，代码中 bmp 是由 ADT 自动生成的 icon.png 图片资源生成一个位图实例。

```
Bitmap bmp = BitmapFactory.decodeResource(this.getResources(),
R.drawable.icon);
```

下面讲解位图常用的操作函数：

1．绘制位图

```
public void myDraw() {
      ...
      canvas.drawBitmap(bmp, 0, 0, paint);
      ...
}
```

代码中使用的 drawBitmap 函数说明如下。

M drawBitmap（Bitmap bitmap, float left, float top, Paint paint)

作用：在画布上绘制一张位图

第一个参数：位图实例

第二、三个参数：位图的 X，Y 坐标

第四个参数：画笔实例

代码执行效果如图 4-14 所示。

2．旋转位图

```
public void myDraw() {
      ...
      canvas.rotate(30, bmp.getWidth()/2, bmp.getHeight()/2);
      canvas.drawBitmap(bmp, 0, 0, paint);
      ...
}
```



图 4-14 绘制位图

代码中使用的 rotate 函数说明如下。

M rotate（float degrees, float px, float py）

作用：旋转画布

第一个参数：画布旋转的角度

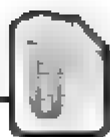
第二、三个参数：画布的旋转点

如果旋转的角度大于 0，顺时针旋转；旋转的角度小于 0，则逆时针旋转。

代码执行效果如图 4-15 所示。



图 4-15 旋转画布



提示

Canvas 中旋转画布还有一个函数：rotate（float degrees），参数传入的是旋转画布的角度，此种方法无法设置旋转点，默认旋转点为屏幕的中心点。

如果想让位图进行旋转，那么通过这种旋转画布的方法即可实现。但是当使用此种实现画布旋转时，要注意两点：

- 如果希望图片的旋转是以图片中心点进行旋转，那么在使用 rotate 旋转画布函数对画布进行旋转时，其旋转点坐标应该设置为图片的中心点坐标；

- rotate 函数是对整个画布进行旋转操作，也就是意味着，画布上所有绘制的元素都会因画布的旋转而进行对应的旋转。

例如代码修改为：

```
public void myDraw() {
    ...
    canvas.rotate(30, bmp.getWidth()/2, bmp.getHeight()/2);
    canvas.drawBitmap(bmp, 0, 0, paint);
    canvas.drawBitmap(bmp, 100, 0, paint);
    ...
}
```

当旋转画布后，绘制两张位图，观察如图 4-16 所示的效果。



图 4-16　整个画布进行旋转

通过图 4-16 可以明显看出，第二次绘制的位图也被旋转了！当然这并不是想要效果，那么如果只想对一张位图进行旋转操作该如何实现呢？

这就需要大家熟悉 Canvas 类中两个很重要的函数 save() 与 restore():

- save()：作用是用于保存当前画布的状态；
- restore()：作用是恢复上次保存的画布状态。

这两个函数是配对出现的，也就是说有 N 个 save()函数出现，必须有 N 个 restore()函数对应出现；当然 restore()函数的出现次数可以大于 save()函数，但是绝对不能让 save()函数出现的次数大于 restore()函数。

既然有对画布状态进行保存和状态恢复的函数，那么就可以单独对一张位图进行旋转操作了，针对一张位图进行旋转操作的步骤如下：

对画布进行旋转之前，首先利用画布的 save()函数将其画布的状态保存起来，然后对画布进行旋转，紧接着绘制位图，最后当位图绘制完后将画布状态恢复。

这样一来，不管画布以后再进行绘制位图、图形、还是其他操作都不会受到影响了。

对位图旋转代码进行修改：

```
public void myDraw() {
```

```
    ...
    canvas.save();
    canvas.rotate(30, bmp.getWidth()/2, bmp.getHeight()/2);
    canvas.drawBitmap(bmp, 0, 0, paint);
    canvas.restore();
    canvas.drawBitmap(bmp, 100, 0, paint);
    ...
}
```

项目运行效果如图 4-17 所示。



图 4-17　保存画布状态

从图 4-17 所示的效果可以看到第二张位图的绘制并没有因第一张位图绘制前，对画布进行旋转而受到任何的影响，这全归功于 save()与 restore()两个函数！

其实不光是 Canvas（画布类）中的旋转函数会对整个画布进行操作，还有画布的缩放、画布的位移也都是对整个画布进行操作，所以大家一定要牢记：当对画布进行缩放、旋转和位移操作时，为了保证其他绘制的元素不受影响，应该利用 save()与 restore()对画布进行适当的保存与恢复操作。

以上介绍的位图旋转，是通过对画布进行操作实现的。除此之外还有一种对位图进行旋转的方式，就是利用矩阵 Matrix 来实现如图 4-15 的效果，代码如下所示：

```
public void myDraw() {
    ...
    Matrix mx = new Matrix();
    mx.postRotate(30, bmp.getWidth() / 2, bmp.getHeight() / 2);
    canvas.drawBitmap(bmp, mx, paint);
    ...
}
```

首先创建一个矩阵实例，然后对矩阵进行旋转缩放，最后在使用画布绘制位图时，将矩形信息作为参数传入即可。

Matrix 类的 postRotate 函数与 Canvas 中的 rotate 函数的作用相同，参数表示的函数也都

一致。

使用矩阵对位图进行操作时，可以免去对画布的状态保存和恢复，因为矩阵就是针对单独位图进行的操作，所以不会影响画布其他元素的绘制；当然这里所说的对位图的操作不仅仅是旋转，在后续讲解的位图缩放和位图位移都可以利用矩阵来实现。

3．平移位图

```
public void myDraw() {
        ...
        canvas.save();
        canvas.translate(10, 10);
        canvas.drawBitmap(bmp, 0, 0, paint);
        canvas.restore();
        ...
}
```

代码中使用的 translate 函数说明如下。

M translate（float dx, float dy）

作用：平移画布

第一个参数：在 X 轴上平移画布距离

第二个参数：在 Y 轴上平移画布距离

不管是在 X 轴还是在 Y 轴上进行平移，其平移的距离值大于 0 时，则表示向 X 或 Y 轴的正方向进行平移；当平移的距离值小于 0 时，则表示向 X 或 Y 轴的负方向进行平移。

项目运行效果如图 4-18 所示。



图 4-18　平移画布

当然这里对位图进行平移也只是利用画布的平移来实现。我们在讲解位图旋转时提到过，利用矩阵也可以完成对位图平移的操作。

利用矩阵对位图进行平移：

```
public void myDraw() {
```

```
    ...
    Matrix maT = new Matrix();
    maT.postTranslate(10, 10);
    canvas.drawBitmap(bmp, maT, paint);
    ...
}
```

矩阵的 postTranslate 方法与画布的 translate 方法雷同，这里不再赘述。

4．缩放位图

```
public void myDraw() {
    ...
    canvas.save();
    canvas.scale(2f, 2f, 50 + bmp.getWidth() / 2, 50 +
                      bmp.getHeight() / 2);
    canvas.drawBitmap(bmp, 50, 50, paint);
    canvas.restore();
    canvas.drawBitmap(bmp, 50, 50, paint);
    ...
}
```

代码中使用的 scale 函数说明如下。

M scale（float sx, float sy, float px, float py）

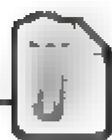
作用：对画布进行缩放

第一个参数：对画布 X 轴的缩放比例

第二个参数：对画布 Y 轴的缩放比例

第三、四个参数：对画布缩放的起始点

第一、二个参数表示的 X,Y 缩放的比例是相对于缩放起始点进行的，分别表示 X,Y 轴缩放的比例值；坐标轴的比例值只要都为 1 时表示画布没有进行缩放；当比例值大于 1 时表示放大；当比例值小于 1 且大于 0 时表示缩小。



提示

Canvas中表示缩放画布的还有一个函数：scale（float sx, float sy），两个参数表示缩放画布X与Y轴的比例值；此种方法无法设置缩放起始点，默认缩放起始点为屏幕的（0,0）点。

项目运行效果如图 4-19 所示。

这里再强调一下，不管是画布的旋转还是缩放，如果想让某一位图以位图的中心点进行旋转或缩放，那么旋转的旋转点或缩放的起始点都应该设置为，在绘制位图的 X、Y 的坐标的基础上分别再加上位图宽的一半与位图高的一半。

- 绘制位图都是默认从位图的左上角进行绘制；

● 对于一个规则的位图来说，当用其位图的 X 坐标加上位图宽的一半，用其位图的 Y 坐标加上位图高的一半，得到的两个坐标值就是位图的中心点。



图 4-19　画布的缩放

利用矩阵对位图进行缩放：

```
public void myDraw() {
        ...
        //X 轴镜像
        canvas.save();
        canvas.scale(-1, 1, bmp.getWidth() / 2, bmp.getHeight() / 2);
        canvas.drawBitmap(bmp, 0, 0, paint);
        canvas.restore();
        //Y 轴镜像
        canvas.save();
        canvas.scale(1, -1, bmp.getWidth() / 2, bmp.getHeight() / 2);
        canvas.drawBitmap(bmp, 0, 0, paint);
        canvas.restore();
        ...
}
```

这里利用矩阵对位图缩放时，除了对矩阵进行缩放外还对矩阵进行了一步平移操作。

其实细心的大家可能早就观察出来了，利用矩阵对位图进行操作，存在一个缺点就是位图无法设置其位置，都是默认放置在屏幕的（0,0）点。

例如有一张位图，想将位图绘制在（60,60）这里，然后再对位图进行缩放、平移与旋转操作。首先都知道对位图的操作可以完全利用矩阵来实现，但是在绘制带矩阵信息的位图方法中：

```
drawBitmap(Bitmap bitmap, Matrix matrix, Paint paint)
```

此方法是无法设置图片的位置的，利用此种带矩阵的绘制位图函数，其位图的坐标信息

放在了矩阵里，而矩阵本身并没有设置初始位置的函数，此时只能利用矩阵的平移函数来弥补这个问题。所以利用矩阵进行操作位图时，当位图不想默认放在画布（0,0）点时候，需要通过矩阵的 postTranslate 函数进行位图位置的设置。

5．镜像反转位图

```
public void myDraw() {
        ...
        //X 轴镜像
        canvas.save();
        canvas.scale(-1, 1, 100 + bmp.getWidth() / 2, 100 +
                            bmp.getHeight() / 2);
        canvas.drawBitmap(bmp, 100, 100, paint);
        canvas.restore();
        //Y 轴镜像
        canvas.save();
        canvas.scale(1, -1, 100 + bmp.getWidth() / 2, 100 +
                            bmp.getHeight() / 2);
        canvas.drawBitmap(bmp, 100, 100, paint);
        canvas.restore();
        ...
}
```

对位图的镜像反转，利用画布对位图的缩放函数来实现；在之前讲解缩放位图时，对 scale（float sx, float sy, float px, float py）函数详细介绍过其四个参数表示的含义，但是对其中第一、二个参数表示 X、Y 轴的缩放比例值的范围只是讲解了 X、Y 值都大于 0 的情况。当 X 轴的比例值小于 0 时，其实是对画布进行 X 轴的镜像后的缩放；而当 Y 轴的比例值小于 0 时，是对画布进行 Y 轴镜像后的缩放。也就是说，当 X 或者 Y 轴的缩放比例值小于 0 并且大于-1 时，是对画布进行 X 或 Y 轴镜像后的缩小操作；当 X 或者 Y 轴的缩放比例值小于-1 时，是对画布进行 X、Y 轴镜像后的放大操作。所以如果只是想让位图沿着 X 轴或者 Y 轴进行镜像操作，那么 X 与 Y 的缩放比例都应该设置为-1，保证位图镜像不被缩放。

因此，对位图进行 X 轴的镜像操作，只需要设置 X 轴的缩放比例为-1 即可。项目运行效果如图 4-20 所示。

对位图进行 Y 轴的镜像操作，只需要设置 Y 轴的缩放比例为-1 即可。项目运行效果如图 4-21 所示。



图 4-20　X 轴镜像反转位图



图 4-21　Y 轴镜像反转位图

利用矩阵实现对位图的镜像操作代码如下：

```
public void myDraw() {
    ...
    //X 轴镜像
    canvas.drawBitmap(bmp, 0, 0, paint);
    Matrix maMiX = new Matrix();
    maMiX.postTranslate(100, 100);
    maMiX.postScale(-1, 1, 100 + bmp.getWidth() / 2, 100 +
                          bmp.getHeight() / 2);
    canvas.drawBitmap(bmp, maMiX, paint);
    //Y 轴镜像
    canvas.drawBitmap(bmp, 0, 0, paint);
    Matrix maMiY = new Matrix();
```

```
        maMiY.postTranslate(100, 100);
        maMiY.postScale(1, -1, 100 + bmp.getWidth() / 2, 100 +
                                  bmp.getHeight() / 2);
        canvas.drawBitmap(bmp, maMiY, paint);
        ...
}
```

在介绍位图时，提到过 Android 系统几种支持的图片格式。在游戏开发中，一般最常用的是 png 格式的图片，原因在于 png 格式的图片支持透明度。

下面通过举例来更形象的阐述常用 png 的原因。

如下有两张大小、内容相同的图片，只是左侧的位图是由一张 png 的半透明图片创建而成，而右侧的位图是由 jpg 格式的图片创建而成，如图 4-22 所示。



图 4-22　绘制 png 或 jpg 的位图

通过画布的黑色背景衬托，右侧 jpg 格式的图片的白色背景很明显地显现出来；而左边是 png 的半透明图，背景为透明，图上的像素也全部半透明。

大家都知道画布在绘制位图、图形、文本等对象时，其覆盖关系都是按照先后顺序。比如在画布上绘制两张相同规格与格式的位图，首先绘制了位图 bmp1，然后在相同的位置绘制 bmp2，那么 bmp2 会覆盖 bmp1。

下面通过一段代码更好地解释 png 的特性：bmpPng 是对应 png 格式的位图，bmpJpg 是对应 jpg 格式的位图。

```
public void myDraw() {
        ...
        //左侧圆形与 png 位图
        canvas.drawCircle(30, 15, 10, paint);
        canvas.drawBitmap(bmpPng, 0, 0, paint);
        //右侧圆形与 jpg 位图
        canvas.drawCircle(150, 15, 10, paint);
        canvas.drawBitmap(bmpJpg, 120, 0, paint);
```

```
    ...
}
```

项目运行效果如图 4-23 所示。

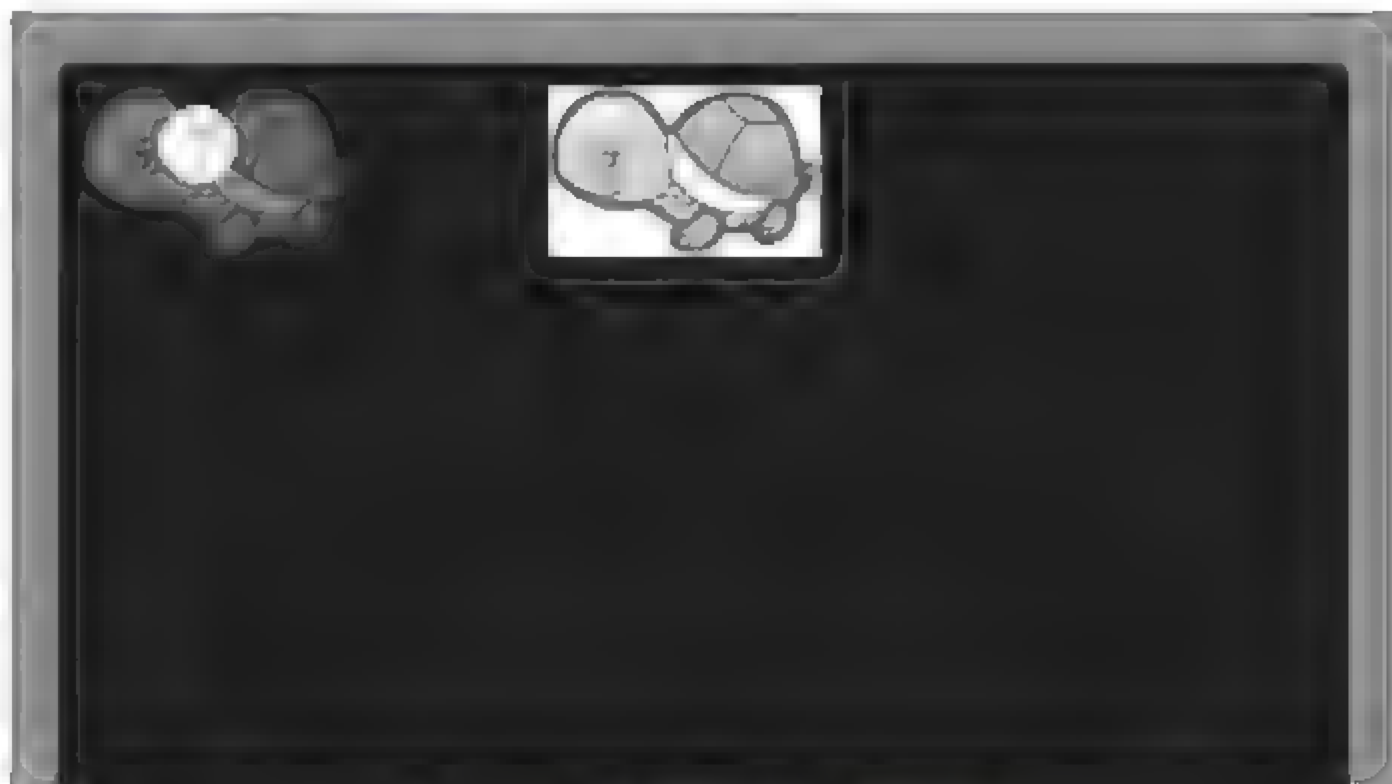


图 4-23　位图覆盖的不同效果

从上面代码中可以很明显地看出，在绘制这两种格式的位图之前首先都绘制一个圆形。通过图 4-23 可以看出，左侧圆形虽然被 png 格式的位图所覆盖，但是因为这张 png 的图是半透明，所以可以透过这张 png 位图隐约看到被覆盖的圆形；而右侧被 jpg 位图覆盖的圆形是无法看到的，原因在于 jpg 格式的图片无法保存透明度，所以也无法透过 jpg 位图看到被其覆盖的内容。

这里虽然简单介绍了 png 图片格式的好处，但不是说在开发游戏中必须使用 png 图。使用什么格式的图片，只能根据具体项目的需求来选择。比如单纯地希望找一张图片作为游戏背景，那么选用 jpg、png 或者 bmp 格式的图片都可以。

4.10 剪切区域

剪切区域在游戏开发中也是画布很常用的一个函数，是游戏开发需要重点掌握的知识点。剪切区域也称为可视区域，是由画布进行设置的；它指的是在画布上设置一块区域，当画布一旦设置了可视区域，那么除此区域以外，绘制的任何内容都将看不到；可视区域可以是个圆形、矩形等等。

新建项目“ClipCanvasProject”，游戏框架使用 SurfaceView 游戏框架，项目对应的源代码为“4-10（可视区域）”。修改 MySurfaceView 类中的绘图函数如下：

```
public void myDraw() {
    ...
    canvas.clipRect(0, 0, 20, 20);
```

```
    canvas.drawRect(0, 0, this.getWidth(), this.getHeight(), paint);
    ...
}
```

其中clipRect（int left, int top, int right, int bottom）函数的作用是为画布设置矩形可视区域。函数第一、二个参数为可视区域的左上角坐标，第三、四个参数为可视区域的右下角坐标。

项目运行效果如图4-24所示。



图4-24　剪切区域

通过上面代码可以看出，要绘制的矩形是个等同于画布大小的矩形，但是从图4-24的效果来看，并不是预期的结果，产生这一现象的原因是剪切区域在发生作用。

代码中可以看到画布在绘制矩形之前设置了一个起始点坐标（0,0），宽20，高20的矩形，而图4-24的效果显示的也正好是一个起点为（0,0），宽高都是20的矩形；也就是说图4-24的效果中所看到的矩形就是画布设置的剪切区域。

这里要注意一点，因为Canvas设置剪切区域的函数也是对整个画布进行操作，所以为了避免画布上其他绘制的元素受到影响，在设置剪切区域前，也应该保存画布状态，并且在处理过后还原画布的状态。因此，以上代码应该修改为：

```
public void myDraw() {
    ...
    canvas.save();
    canvas.clipRect(0, 0, 20, 20);
    canvas.drawRect(0, 0, this.getWidth(),this.getHeight(),paint);
    canvas.restore();
    ...
}
```

为了便于观察，下面对一张图片设置矩形剪切区域。首先将如图 4-25 所示的图片放入项目中。



图 4-25　image.png

首先通过此图片资源生成一张 Bitmap 位图：

```
Bitmap bmp = BitmapFactory.decodeResource(this.getResources(),
R.drawable.image);
```

然后修改绘图函数，为其位图设置矩形可视区域：

```
public void myDraw() {
        ...
        canvas.save();
        canvas.clipRect(0, 0, 20, 20);
        canvas.drawBitmap(bmp, 0, 0, paint);
        canvas.restore();
        ...
}
```

项目运行效果如图 4-26 所示。

图 4-26 所示的左侧是没有设置可视区域直接绘制位图的效果，而右侧则是对画布设置了可视区域的效果，通过此对比图可以明显地看出可视区域的作用。

除了设置矩形可视区域之外，画布还提供了其他两种设置可视区域的方法。

① 第一种是利用 Path 来设置可视区域的形状。

M clipPath（Path path）

作用：为画布设置可视区域

参数：Path 实例

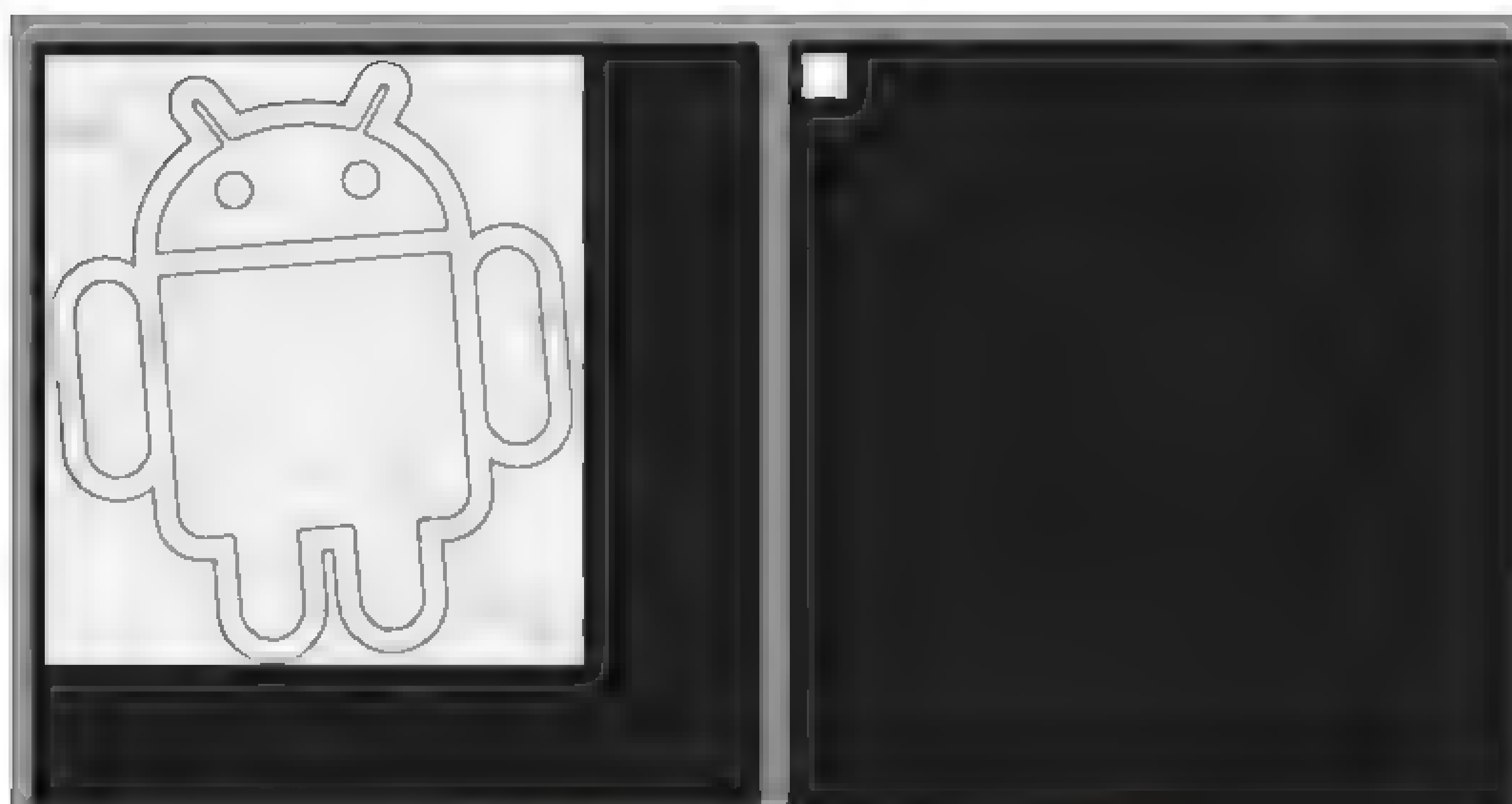


图 4-26 设置可视区域前后对比图

利用此函数，通过 Path 可以为画布设置任何需要的可视区域，下面利用 Path 为位图设置一个圆形可视区域：

```
public void myDraw() {
        ...
        canvas.save();
        Path path = new Path();
        path.addCircle(30, 30, 30, Direction.CCW);
        canvas.clipPath(path);
        canvas.drawBitmap(bmp, 0, 0, paint);
        canvas.restore();
        ...
}
```

项目运行效果如图 4-27 所示。



图 4-27 利用 Path 设置可视区域

②第二种是利用Regin来对画布设置可视区域。

clipRegion（Region region）

作用：为画布设置可视区域

参数：Region 实例

这里简单介绍一下 Region 这个类：Region 表示区域的集合，所以它可以设置多个区域块，而且可以通过这些区域块之间的关系来处理一些问题；比如 Region 设置它所有区域块相交的区域是否可见、设置相交区域只让交集显示等等。

Region 常用的函数：

M op（Rect rect, Op op)

作用：设置区域块

第一个参数：Rect 实例

第二个参数：Region.Op静态值，表示区域块的显示方式。其中区域块的显示方式如下：

- Region.Op.UNION：区域全部显示；
- Region.Op.INTERSECT：区域的交集显示；
- Region.Op.XOR：不显示交集区域。

下面通过 Region 设置画布可视区域：

```
public void myDraw() {
        ...
        canvas.save();
        Region region = new Region();
        region.op(new Rect(20,20,100,100), Region.Op.UNION);
        canvas.clipRegion(region);
        canvas.drawBitmap(bmp, 0, 0, paint);
        canvas.restore();
        ...
}
```

项目运行效果如图 4-28 所示。

在上面代码中，region.op（new Rect(20,20,100,100), Region.Op.UNION）的参数 Region.Op.UNION 表示除了此区域与 Region 中的其他区域的交集部分外都显示；但是其作用仅仅通过一个区域块是无法看出效果的，所以下面在 Region 中再设置一个区域块，修改绘图函数如下：

```
public void myDraw() {
        ...
        canvas.save();
        Region region = new Region();
        region.op(new Rect(20,20,100,100), Region.Op.UNION);
        region.op(new Rect(40,20,80,150), Region.Op.XOR);
```

```
        canvas.clipRegion(region);
        canvas.drawBitmap(bmp, 0, 0, paint);
        canvas.restore();
        ...
}
```



图 4-28 用 Region 设置可视区域

项目运行效果如图 4-29 所示。



图 4-29 Region 区域块

从图 4-29 中可以看出，Region 设置的第一个区域块全部显示，第二个区域块没有显示交集部分。除了利用 Region 实现设置可视区域外，游戏开发中还经常利用 Region 来检测一个点或者一个矩形等是否在与 Region 中的区域块发生碰撞，对于 Region 的这个功能将在后面介绍碰撞检测时再详细讲解。

4.11 动画

一款游戏中必不可少的就是动态的元素，这些动态的元素可能是角色的移动、爆炸的效果、过场的特效等等；那么针对动画的形成，本节将介绍两种方式，一种是系统提供的Animation类的特效，另外一种方式是由自己手动实现。

4.11.1 Animation 动画

在Android中，系统提供了动画类Animation，其中又分为四种动画效果：

- AlphaAnimation：透明度渐变动画；
- ScaleAnimation：渐变尺寸缩放动画；
- TranslateAnimation：移动动画；
- RotateAnimation：旋转动画。

下面首先详细介绍4种动画效果的创建方法。

（1）AlphaAnimation透明度渐变动画

M Animation alphaA = new AlphaAnimation（float fromAlpha, float toAlpha）

第一个函数：动画开始时的透明度

第二个参数：动画结束时的透明度

两个参数的取值范围为[0，1]，从完全透明到完全不透明。

（2）ScaleAnimation渐变尺寸缩放动画

M Animation scaleA = new ScaleAnimation（float fromX, float toX, float fromY, float toY, int pivotXType, float pivotXValue, int pivotYType, float pivotYValue）

第一个参数：动画起始时X坐标上的伸缩比例

第二个参数：动画结束时X坐标上的伸缩比例

第三个参数：动画起始时Y坐标上的伸缩比例

第四个参数：动画结束时Y坐标上的伸缩比例

第五个参数：动画在X轴相对于物体的位置类型

第六个参数：动画相对于物体X坐标的位置

第七个参数：动画在Y轴相对于物体的位置类型

第八个参数：动画相对于物体Y坐标的位置

其中位置类型分为以下三种：

- Animation.ABSOLUTE：相对位置是屏幕左上角，绝对位置；

- Animation.RELATIVE_TO_SELF：相对位置是自身 View，取值为 0 时，表示相对于是自身左上角，取值为 1 是相对于自身的右下角；
- Animation.RELATIVE_TO_PARENT：相对父类 View 的位置。

（3）TranslateAnimation 移动动画

M Animation translateA = new TranslateAnimation（float fromXDelta, float toXDelta, float fromYDelta, float toYDelta）

第一个参数：动画起始时 X 轴上的位置

第二个参数：动画结束时 X 轴上的位置

第三个参数：动画起始时 Y 轴上的位置

第四个参数：动画结束时 Y 轴上的位置

（4）RotateAnimation 旋转动画

M Animation rotateA = new RotateAnimation（float fromDegrees, float toDegrees, int pivotXType, float pivotXValue, int pivotYType, float pivotYValue）

第一个参数：动画起始时的旋转角度

第二个参数：动画旋转到的角度

第三个参数：动画在 X 轴相对于物件位置类型

第四个参数：动画相对于物件的 X 坐标的开始位置

第五个参数：动画在 Y 轴相对于物件位置类型

第六个参数：动画相对于物件的 Y 坐标的开始位置

在 Animation 中的四种动画创建都是 new 出来的，虽然创建很简单，但是根据参数的不同可以构造出千变万化的动画效果。不管哪一种动画，都有一些通用的方法，比如：

- restart()：重新播放动画；
- setDuration（int time）：设置动画播放时间，单位是毫秒。

动画的创建完成之后，接下来就是如何启动动画。其实在 View 视图中，启动动画也是非常的简单，只要调用 View 类的 startAnimation（Animation animation）函数即可。知道了动画的创建与播放，下面就创建一个项目来实战演练一下吧。

新建项目“AnimationProject”，游戏框架为 View 游戏框架，项目对应的源代码为“4-11-1（Animation 动画）”。

首先运行项目，观察四种效果的截图，如图 4-30、图 4-31、图 4-32、图 4-33 所示。



图 4-30　透明渐变动画效果



图 4-31　缩放动画效果

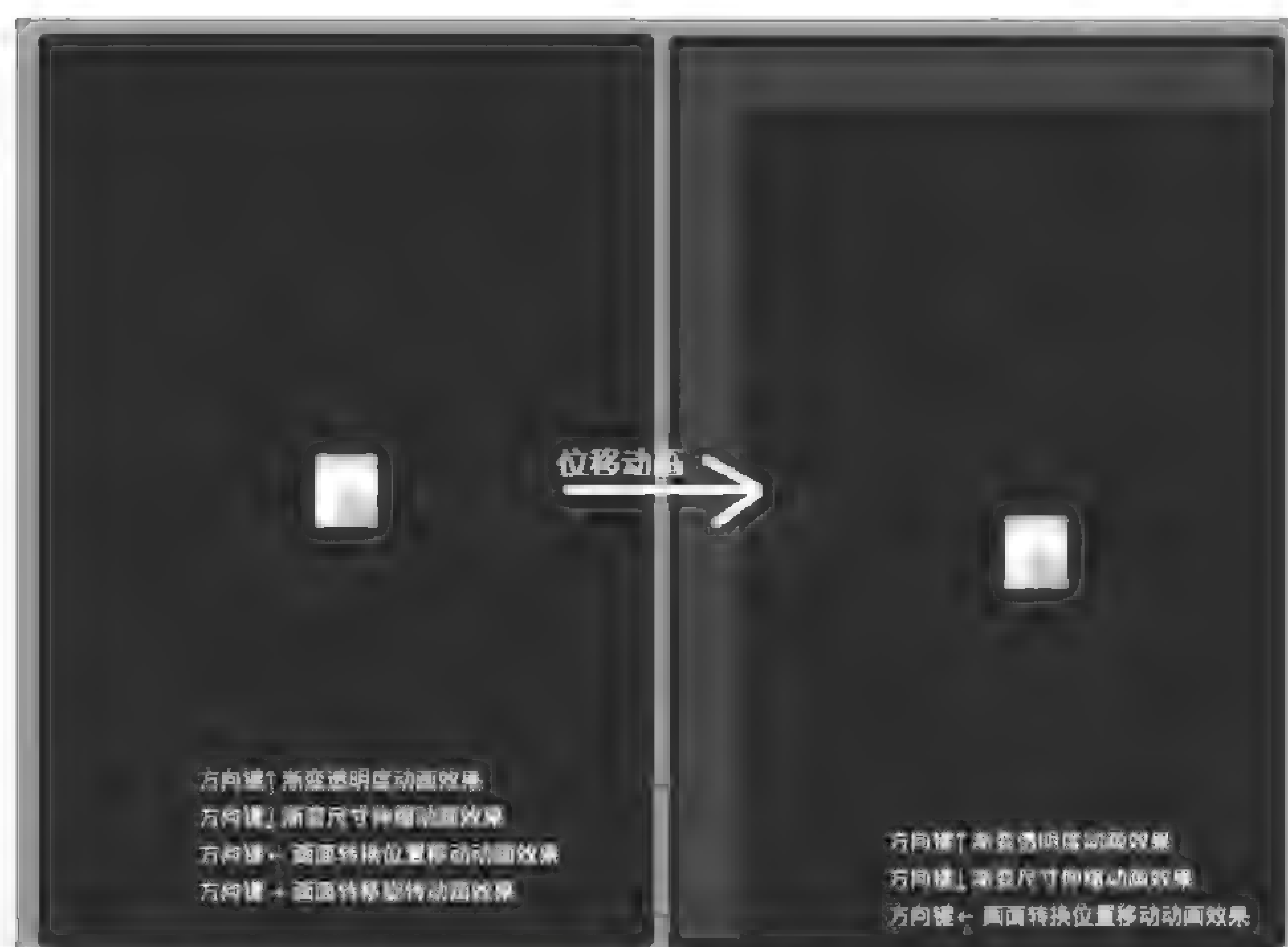


图 4-32　位移动画效果

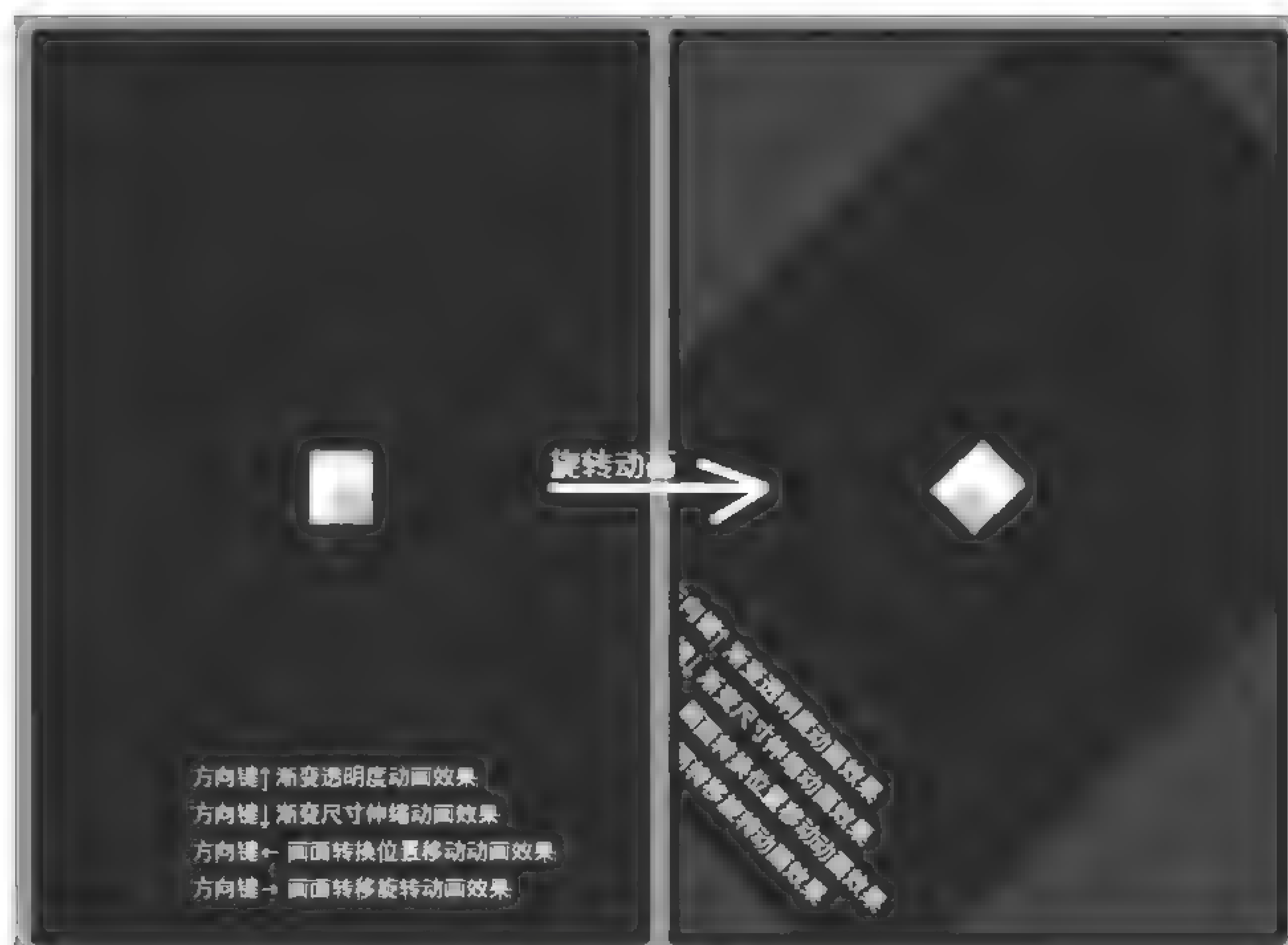


图 4-33　旋转动画效果

在项目代码中，绘图函数简单绘制了一些文本信息和一个 icon 位图。

```
@Override
protected void onDraw(Canvas canvas) {
      super.onDraw(canvas);
      //黑色背景
      canvas.drawColor(Color.BLACK);
      canvas.drawText("方向键↑ 渐变透明度动画效果", 80, this.getHeight() - 80,
                  paint);
      canvas.drawText("方向键↓ 渐变尺寸伸缩动画效果", 80, this.getHeight() -
                  60, paint);
      canvas.drawText("方向键← 画面转换位置移动动画效果", 80, this.getHeight()
                  - 40, paint);
      canvas.drawText("方向键→ 画面转移旋转动画效果", 80, this.getHeight() -
                 20, paint);
      //绘制位图 icon
      canvas.drawBitmap(bmp, this.getWidth() / 2 - bmp.getWidth() / 2,
                   this.getHeight() / 2 - bmp.getHeight() / 2, paint);
                    x += 1;
}
```

项目中每个动画的实例与播放都由不同的四方向按键触发，按键监听事件函数如下：

```
@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
        if (keyCode == KeyEvent.KEYCODE_DPAD_UP) {// 渐变透明度动画效果
             mAlphaAnimation = new AlphaAnimation(0.1f, 1.0f);
             mAlphaAnimation.setDuration(3000);
             // //设置时间持续时间为 3000 毫秒=3 秒
             this.startAnimation(mAlphaAnimation);
        } else if (keyCode == KeyEvent.KEYCODE_DPAD_DOWN) {
            mScaleAnimation = new ScaleAnimation(0.0f, 2.0f, 1.5f, 1.5f,
            Animation.RELATIVE_TO_PARENT, 0.5f, Animation.RELATIVE_TO_
            PARENT, 0.0f);
             mScaleAnimation.setDuration(2000);
             this.startAnimation(mScaleAnimation);
        } else if (keyCode == KeyEvent.KEYCODE_DPAD_LEFT) {
            mTranslateAnimation = new TranslateAnimation(0, 100, 0, 100);
             mTranslateAnimation.setDuration(2000);
             this.startAnimation(mTranslateAnimation);
         } else if (keyCode == KeyEvent.KEYCODE_DPAD_RIGHT) {
            mRotateAnimation = new RotateAnimation(0.0f, 360.0f, Animation.
            RELATIVE_TO_SELF, 0.5f, Animation.RELATIVE_TO_SELF, 0.5f);
             mRotateAnimation.setDuration(3000);
             this.startAnimation(mRotateAnimation);
      }
```

```
        return super.onKeyDown(keyCode, event);
}
```

每个按键对应一种动画，并且每个按键一旦触发，首先实例动画，然后设置动画的播放时间，最后启动此动画即可。这里唯一要提醒的是，Animation 的每种动画都是对整个画布进行操作。

View 不光提供了这四种特效，也对特效提供了监听器；为动画设置监听的步骤如下：

步骤1 首先使用 android.view.animation.Animation.AnimationListener 接口，或者内部类实现此接口；

步骤2 然后重写接口的 3 个抽象函数:

```
@Override
public void onAnimationStart(Animation animation) {
        //动画开始时响应的函数
}
@Override
public void onAnimationEnd(Animation animation) {
        //动画结束时响应的函数
}
@Override
public void onAnimationRepeat(Animation animation) {
        //动画重播时响应的函数
}
```

三个函数监听动画的不同状态，其中三个函数的参数都表示当前播放的或者播放结束的动画实例；通过此参数可以对多个设置监听的动画进行判断和匹配。

步骤3 最后使用动画实例Animation.setAnimationListener（AnimationListener listener）设置动画监听器即可。

4.11.2 自定义动画

1. 动态位图

在介绍游戏框架时，曾在屏幕上让文本字符串跟随玩家手指移动，从而形成一个动态的效果；那么让一张位图形成动态效果也很容易，只要不断改变位图的 X 或者 Y 轴的坐标即可。下面就利用一张位图形成海的波浪效果。

新建项目“BitmapMovie”，游戏框架为 SurfaceView 游戏框架，项目对应的源代码为“4-11-2-1（动态位图）”。首先在项目中放入一张 png 图片，如图 4-34 所示，此图宽为 1388，高为 127。



图 4-34　波浪图片

修改 MySurfaceView 类。

初始化添加:

```
//声明一张波浪位图
private Bitmap bmp;
//声明波浪图的 X,Y 坐标
private int bmpX, bmpY;
```

视图构造函数:

```
public void surfaceCreated(SurfaceHolder holder) {
      ...
      bmp = BitmapFactory.decodeResource(this.getResources(),
              R.drawable.water);
      //让位图初始化 X 坐标正好充满屏幕
      bmpX = -bmp.getWidth()+this.getWidth();
      //让位图绘制在画布的最下方，且图片 Y 坐标正好是（屏幕高-图片高）
      bmpY = this.getHeight() - bmp.getHeight();
      ...
}
```

根据以上设置，图片相对于屏幕的位置关系如图 4-35 所示，图中黑色矩形框表示视图的画布（手机屏幕）。

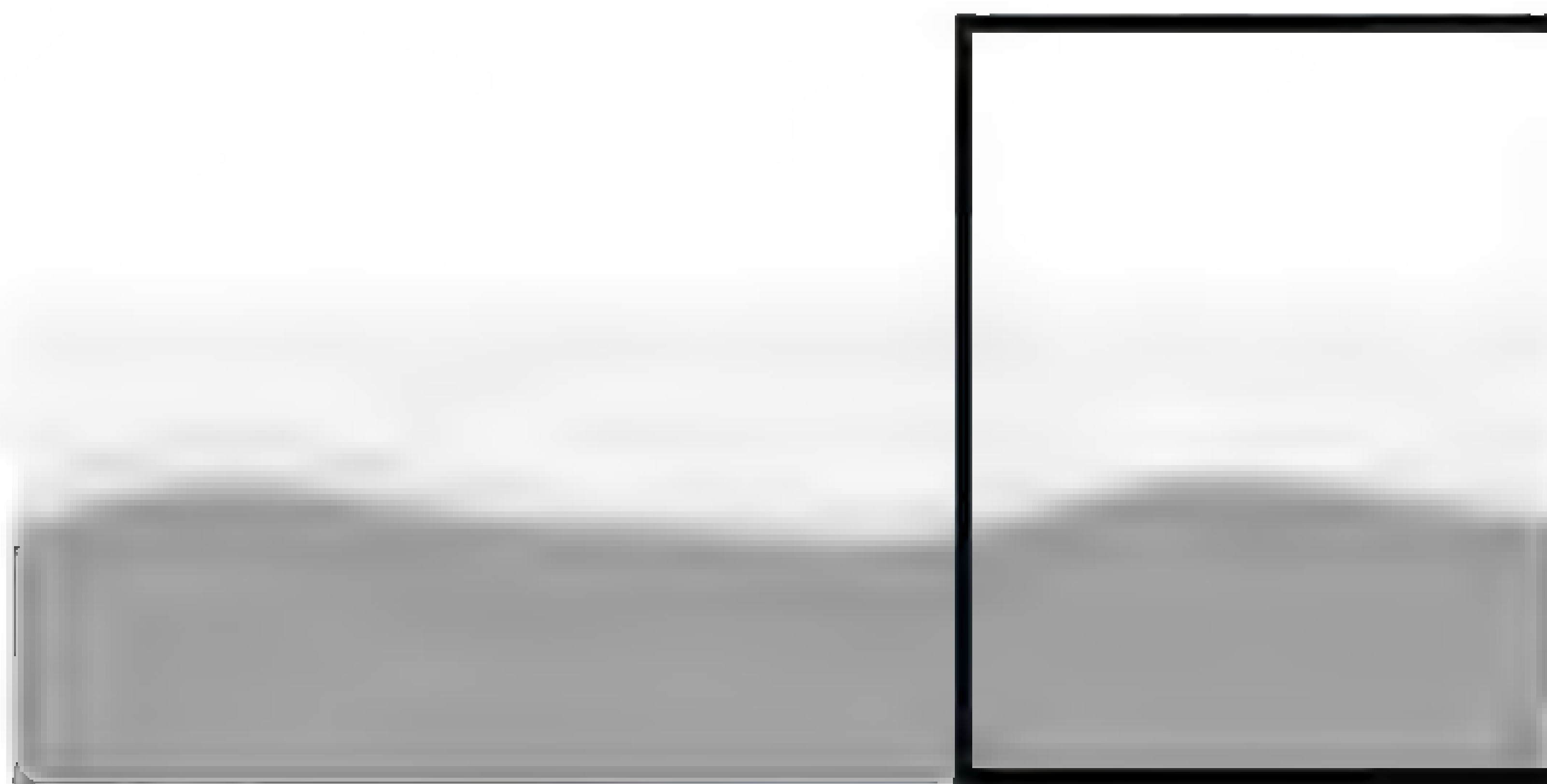


图 4-35　位置对关系图解

这里再次强调，使用 SurfaceView 视图，只有在视图执行完构造函数时才可获取视图的宽高，在此之前获取的屏幕（视图）宽和高都为 0，原因是 SurfaceView 视图还没有创建。

绘制函数:

```
public void myDraw() {
        ...
        canvas.drawBitmap(bmp, bmpX, bmpY, paint);
        ...
}
```

逻辑函数：

```
public void  logic () {
        ...
        bmpX+=5;
        ...
}
```

为了让位图“动”起来，每次重绘屏幕时，让位图的 X 坐标加 5 个像素。这里需要注意，logic()函数在讲述 MySurfaceView 游戏框架时已经介绍过，此函数也可以放在线程中不断地调用执行，用于处理游戏逻辑。

项目运行效果如图 4-36 所示。

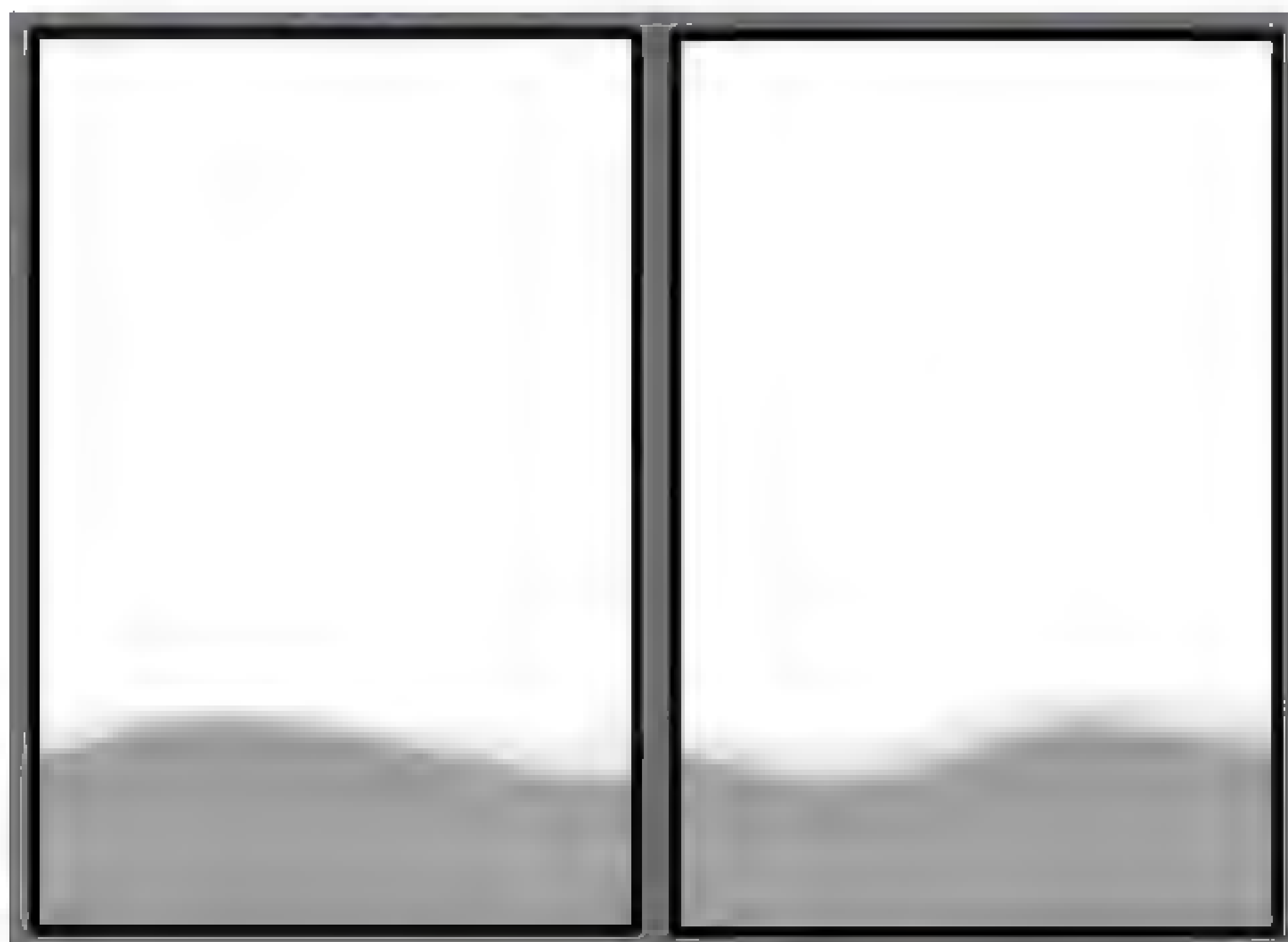


图 4-36　动态位图效果图

2. 帧动画

上一小节是利用改变位图的 X 或者 Y 坐标形成动画效果。当然在游戏开发中，很多动态的帧数不仅仅只有一帧，所以本节讲解如何利用多帧图形成的动画。所谓帧动画，其实就是一帧一帧按照一定的顺序进行播放实现的。

新建项目“FrameMovie”，游戏框架为 SurfaceView 游戏框架，项目对应的源代码为“4-11-2-2（帧动画）”。在项目中导入 10 张 png 图，这十张图片正好是一个动态小鱼的所有帧数，如图 4-37 所示。



图 4-37 10 张小鱼的 png 图

接下来修改 MySurfaceView 类。首先声明一个位图数组，用于存放小鱼的全部帧，并声明一个当前帧的 int 变量，用于操作当前显示帧。

```
Bitmap fishBmp[] = new Bitmap[10];
int currentFrame;
```

在构造函数中，使用 for 循环将小鱼的每个帧存放入帧数组：

```
public MySurfaceView(Context context) {
      ...
      for (int i = 0; i < fishBmp.length; i++) {
          fishBmp[i] = BitmapFactory.decodeResource(this.getResources(),
          R.drawable.fish0 + i); }
      ...
}
```

如果多个资源文件名称按照一定规律命名的话，在 R 资源文件对应生成的 ID 也会有一定规律可循。

绘图函数：

```
public void myDraw() {
      ...
      canvas.drawBitmap(fishBmp[currentFrame], 0, 0, paint);
      ...
}
```

逻辑函数：

```
public void  logic () {
      ...
      currentFrame++;
      if (currentFrame >= fishBmp.length) {
```

```
            currentFrame = 0;
        }
        ...
}
```

为了让动画循环播放，所以对当前帧需要进行控制，当小鱼播放的当前帧大于并且等于小鱼帧数组时，重置当前帧为0（小鱼的第0帧图）。

项目运行效果如图4-38所示。

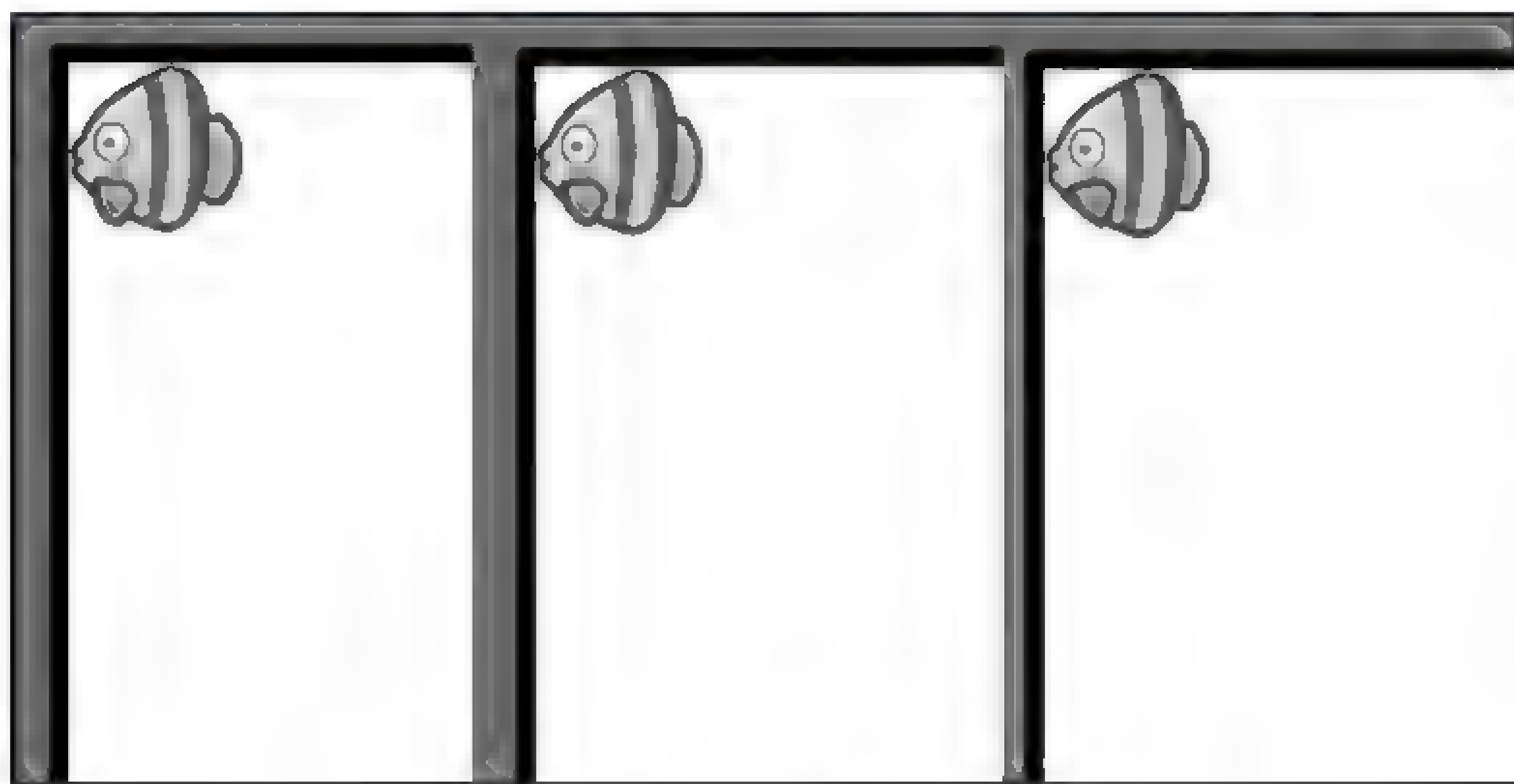


图4-38 帧动画

3．剪切图动画

以上介绍了两种动态效果，这里再介绍一种游戏中最常用的实现动态效果的方式——剪切图动画。剪切图动画类似于帧动画的形式，唯一的区别就是动态物体的动作帧全部放在了一张图片中，然后再通过设置可视区域完成。剪切图动画如图4-39所示。

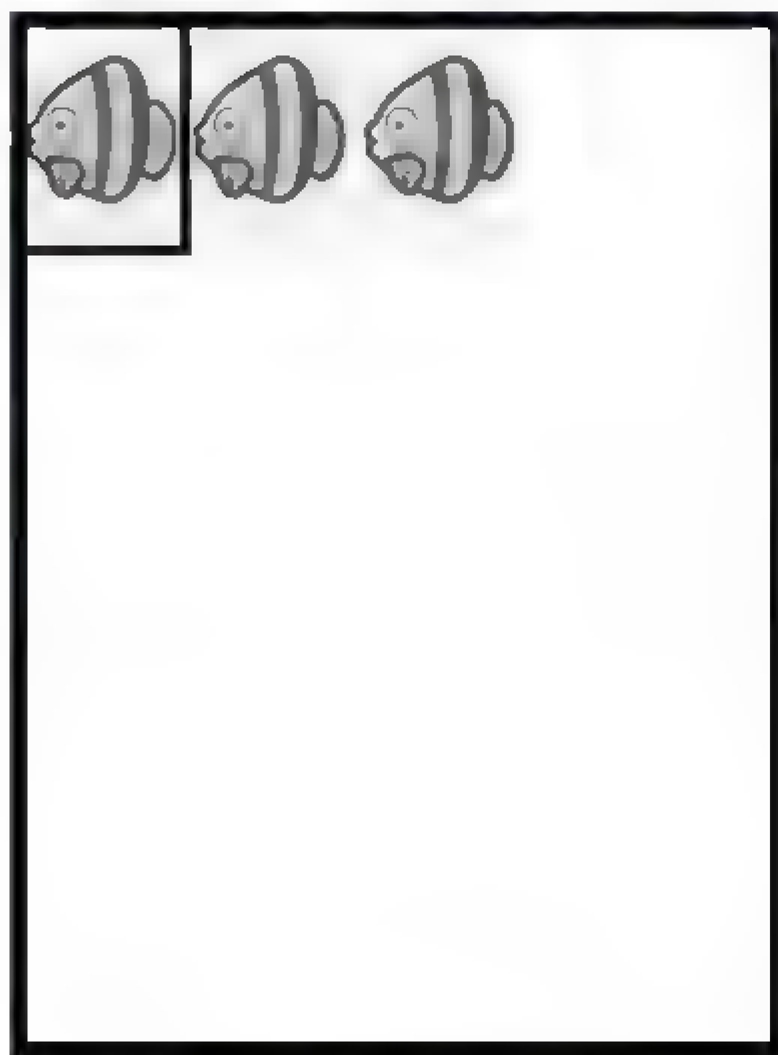


图4-39 演示效果图1

上图中，外层较大的矩形是画布（手机屏幕），小矩形是可视区域。图中可以看到有一张拥有 3 帧的位图，绘制到画布的（0,0）点，并且画布设置了一个与每帧大小相同的可视区域，因此画布只能看到此位图的第一帧。那么，当下次重绘画布时，将位图往 X 轴的反方向移动一帧的宽度距离，如图 4-40 所示。

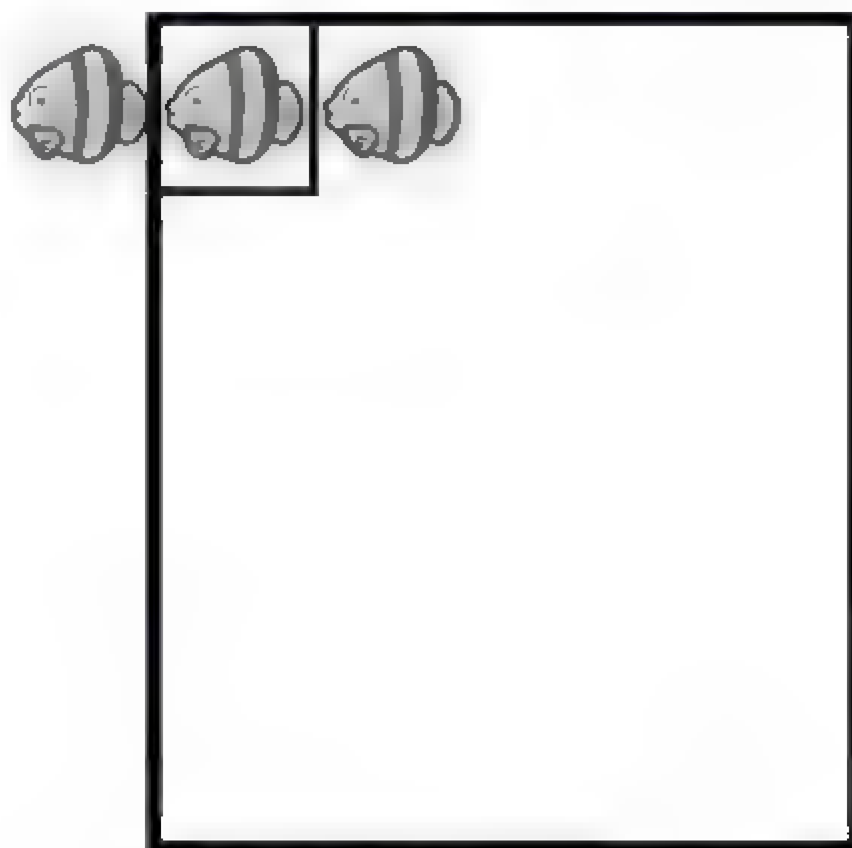


图 4-40　演示效果图 2

此时画布由于设置了可视区域的缘故，只会显示第二帧的区域，那么按照此种方式就可以让设置的可视区域显示位图的每一帧，从而形成动画效果。下面来看一个剪切图动画的范例。

新建项目“ClipBitmapMovie”，游戏框架为 SurfaceView 游戏框架，对应的源代码为“4-11-2-3（剪切图动画）”。首先在项目中导入一张 png 图，如图 4-41 所示。

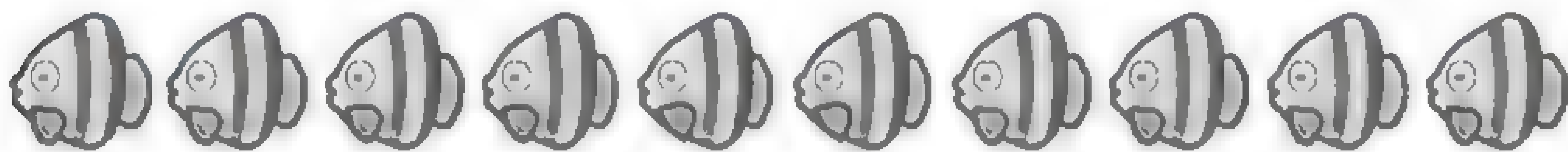


图 4-41　fish.png

接下来创建位图实例，定义当前帧，并实现绘图函数：

```
Bitmap bmpClipBmp = BitmapFactory.decodeResource(this.getResources(),
R.drawable.fish);
int cureentFrame;

public  void  myDraw () {
      ...
      canvas.save();
      //设置画布可视区域（大小是每帧的大小）
      canvas.clipRect(0, 0, bmpClipBmp.getWidth() / 10,
                      bmpClipBmp.getHeight());
      //绘制位图
      canvas.drawBitmap(bmpClipBmp, -cureentFrame *
```

```
                     bmpClipBmp.getWidth() / 10, 0, paint);
        canvas.restore();
        ...
}
```

绘制位图，上面代码默认绘制在画布的（0,0）点，假设将位图绘制在（x,y）点，代码应修改如下：

```
public  void  myDraw () {
        ...
        canvas.save();
        //设置画布可视区域（大小是每帧的大小）
        canvas.clipRect(x, y, x+bmpClipBmp.getWidth() / 10,
                        y+bmpClipBmp.getHeight());
        //绘制位图
        canvas.drawBitmap(bmpClipBmp, x-cureentFrame *
                        bmpClipBmp.getWidth() / 10, y, paint);
        canvas.restore();
        ...
}
```

在绘制位图时，位图的 X 坐标应该减去当前帧乘上帧的宽度，这里因为小鱼的所有帧数高度相同，所以不需要 Y 轴减去当前帧数乘上帧的高度。

逻辑函数代码如下：

```
public void  logic () {
        ...
        cureentFrame++;
        if(cureentFrame>=10){
                cureentFrame=0;
        }
        ...
}
```

逻辑中一直让当前帧不断循环变化，这样每次重绘画布都会显示不同的帧，从而达到动态效果。

4.12 游戏适屏的简述与作用

由于市面上安装 Android 系统的机型不断增多，出现了各种分辨率、各种屏幕尺寸的 Android 系统手机。这种情况下，一个游戏或者一个软件能否在所有的 Android 手机上正常显

示呢？

注意，这里说的是能否在不同 Andriod 手机上正常显示，而不是正常运行!因为所有 Android 应用只要不牵扯到 SDK 版本的功能限制，都能正常安装和运行。这里所说的正常显示的含义举例说明如下：

例如在 4.11.2 小节中完成了一个动态的位图效果，当初始化波浪位图的 Y 坐标时是根据视图的宽高来进行设置的。这种做法的好处是：不管在任意宽高的屏幕上运行显示时都能自适应屏幕。

假设把 4.11.2 小节动态位图项目中位图的初始化代码：

```
bmpY = this.getHeight() - bmp.getHeight();
```

修改为：

```
bmpY = 480 - 127;
```

这里的 480 是视图的高，127 是波浪图片的高。

正常情况下以上两句代码得到的 bmpY 值都是相同的，没有任何问题。但是，如果玩家突然将手机横屏操作呢？效果如图 4-42 所示。

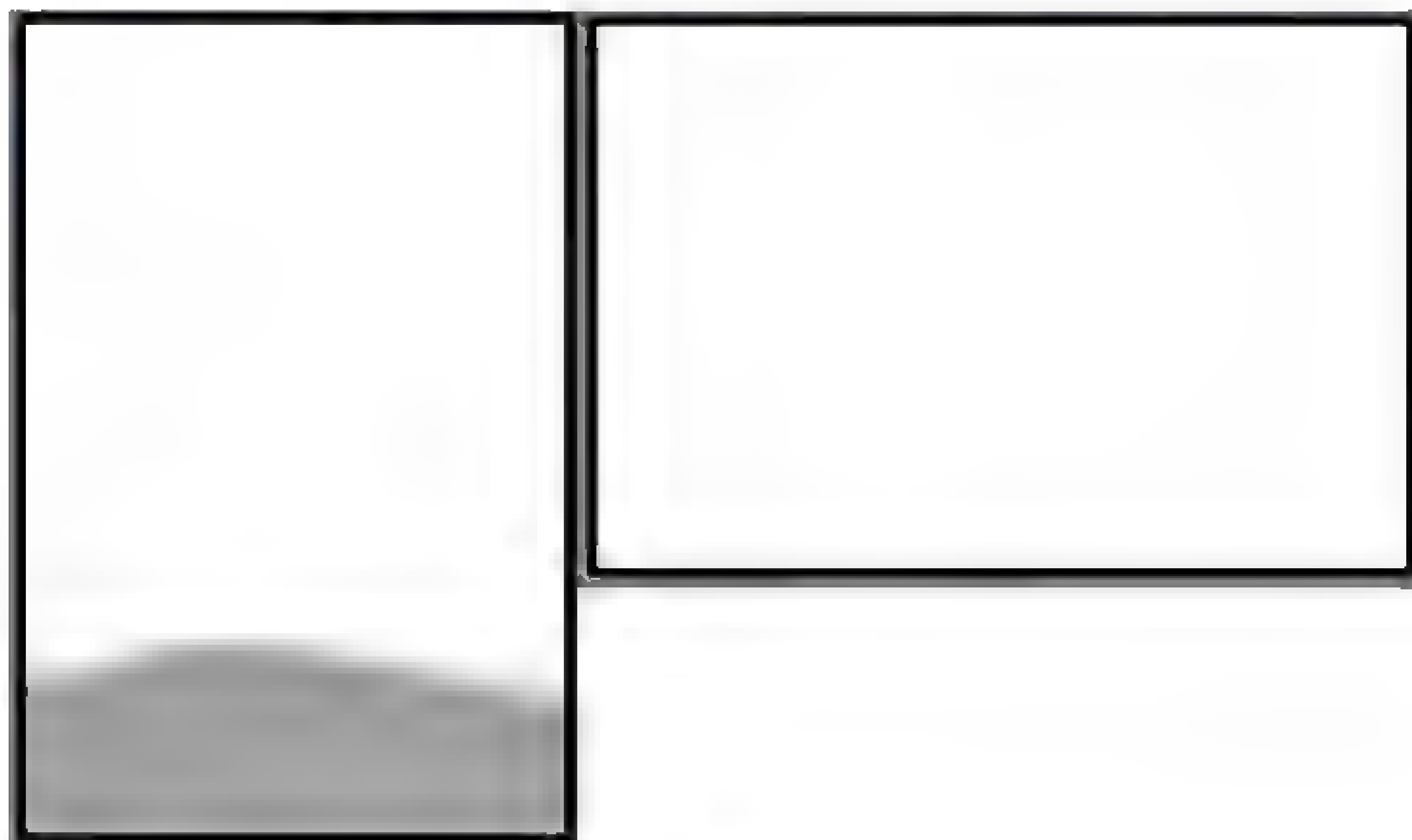


图 4-42　横竖屏对比图 1

通过图 4-42 可以明显看出，当竖屏运行项目时没有任何的问题，但是一旦使用横屏运行项目，则在画布上根本看不到波浪位图。原因很简单，语句“bmpY = 480 – 127;”中，bmpY 是个固定的坐标 353，但是当前使用的模拟器的宽和高在竖屏情况下为：宽=320，高=480；横屏情况下：宽=480，高=320。

通过以上分析可以明显看出，在竖屏下 bmpY 的值超过了屏幕的高度，所以看不到了。但是如果 bmpY 通过屏幕宽高来设定的话：

```
bmpY = this.getHeight() - bmp.getHeight();
```

效果如图 4-43 所示。

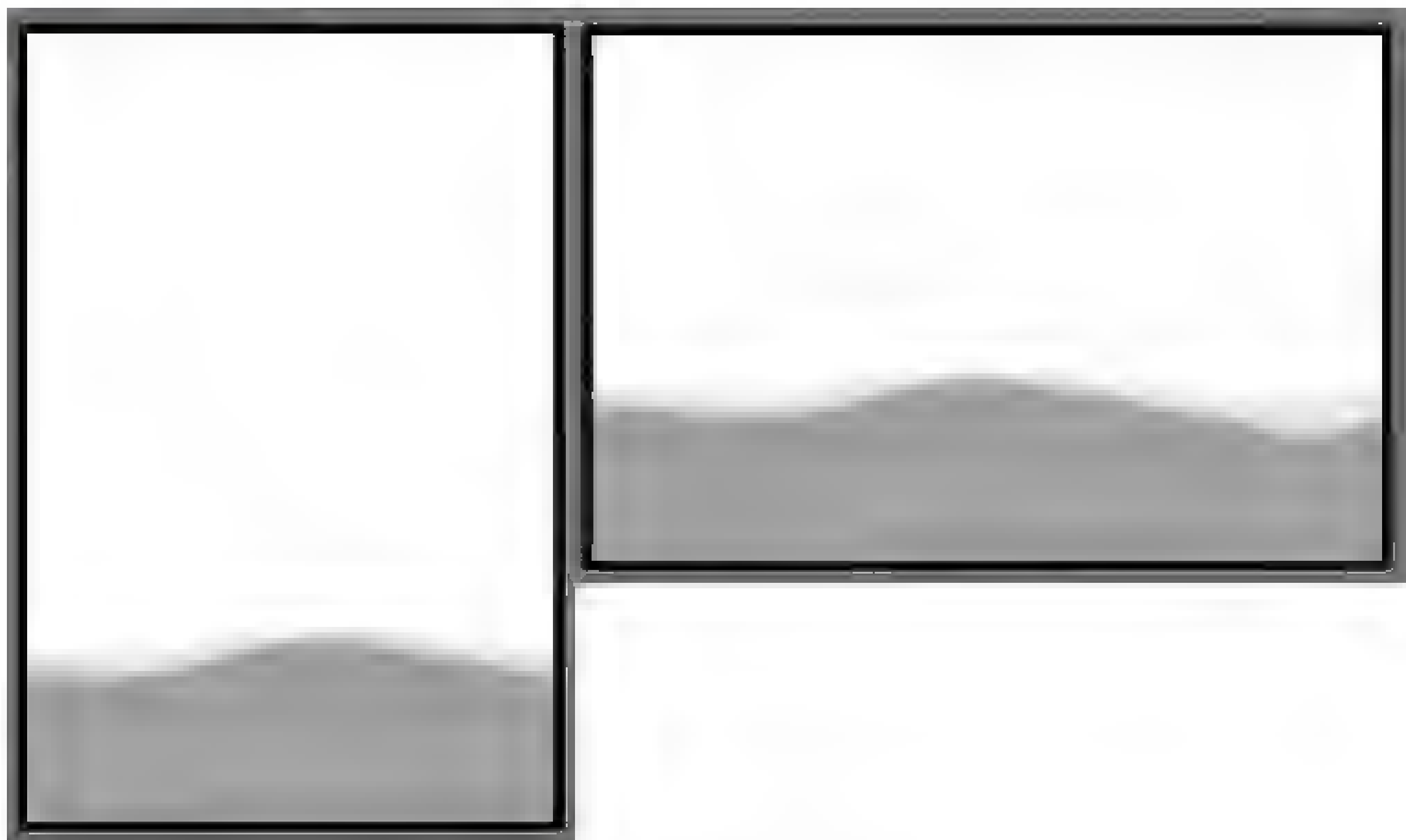


图 4-43 横竖屏对比图 2

有些开发人员可能会想到利用设置禁止横屏或者竖屏的方式来解决，但是在不同分辨率的手机屏幕上仍旧没有从根本上解决问题。其实，这里只是拿手机的横竖屏造成画布宽高的不同来诠释不同分辨率的手机机型。

利用视图宽高来设置位图坐标的好处是：视图的宽高值可以适应不同分辨率的机型。所以，如果想让应用适应更多的 Android 手机，那就要多多使用适屏的做法来开发，这样能节省机型之间移植的时间，毕竟一款手机应用适应的机型越多利益才会越大。

常用的适屏做法有：利用屏幕宽高、位图宽高来设置一些游戏元素的位置；字体的适屏做法最好的是使用字体图，这样文字不会因手机分辨率不同而不同，毕竟图片大小是固定不变的。

4.13 让游戏主角动起来

通过前面小节的学习，已经对位图绘制、操作以及画布、画笔的常用函数有了一定了解和掌握；此小节将讲述如何控制一个游戏主角的移动。

此小节一方面为大家讲解如何将一张由多行多列的动作帧组成的图片实现动态效果；另一方面讲解游戏中那些与玩家直接交互的可控对象，在开发时应该注意的一些知识要点与细节处理。

新建项目“PlayerProject”，游戏框架为 SurfaceView 游戏框架，项目对应的源代码“4-13（操作游戏主角）”。首先在项目中导入一张资源图片，如图 4-44 所示。

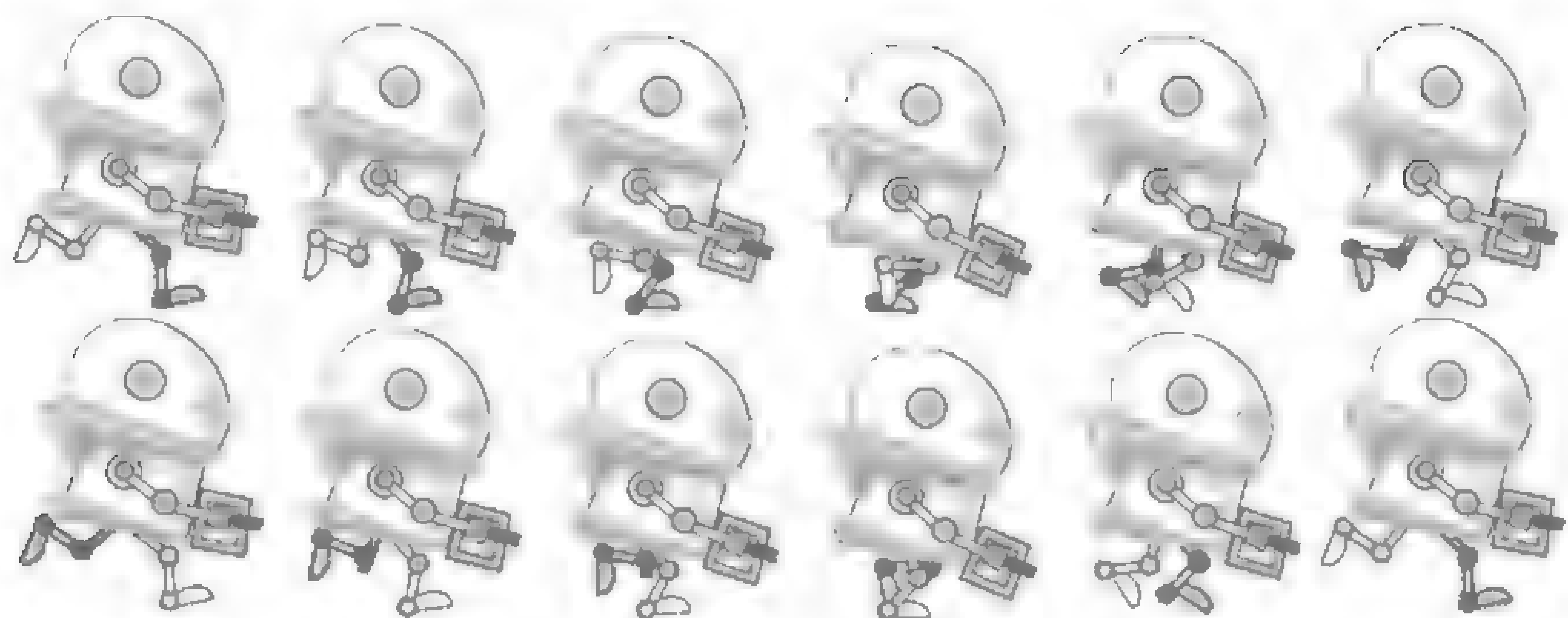


图 4-44 robot.png

此图是机器人向右奔跑的所有动作帧，每帧的大小相同，顺序从上往下，从左往右。修改 MySurfaceView 类如下。

首先声明用到的一些变量：

```
    //机器人位图
    Bitmap bmpRobot= BitmapFactory.decodeResource(this.getResources(),
              R.drawable.robot);
    //机器人的方向常量
    final int DIR_LEFT = 0;
    final int DIR_RIGHT = 1;
    //机器人当前的方向(默认朝右方向)
    int dir = DIR_RIGHT;
    //动作帧下标
    int currentFrame;
    //机器人的X,Y位置
    int robot_x, robot_y;
```

绘图函数：

```
public  void  myDraw () {
    ...
    drawFrame(currentFrame, canvas, paint);
    ...
}
```

在绘图函数中，将绘制的操作都封装在了一个 drawFrame()方法中，有时为了便于观察和修改代码，会有针对性地进行一些函数包装，当然一般这样做是为了增加代码的可复用性。而且一个函数的代码量过大的话，也会造成程序异常！

在观察 drawFrame 函数之前，首先为大家讲解一个知识点。从图 4-44 中可以看到，机器人的动作帧不仅都放在了一张位图上，并且动作帧的十二帧还分成了 6×2 的形式排列，那么

如何利用可视区域来显示每一帧呢？

其实现的方式类似于之前讲解过的“剪切图动画”的处理方式，首先观察一下图4-45。

0	1	2	3	4	5
6	7	8	9	10	11

图4-45 robot.png演示图

图4-45是robot.png图片的示意图，图中数字表示的是动作帧的下标序列，从0帧开始算起，十二帧对应的下标为11。

假设画布设置了一个可视区域，此可视区域的大小与每一帧的大小相同，并且可视区域的位置与第0帧的位置相同。位图的宽高设为：bmpW、bmpY，每帧的宽高设为：frameW、frameH，每一帧的位置相对于整个位图的坐标位置设为：frameX、frameY。

那么通过图片的宽高与每帧的宽高可以得到所有的帧在这个位图上被分为了几行几列。此时，列数为col（col=bmpW/frameW）。

通过以上设置的变量可以求出每一帧相对于整个位图的坐标：

- 第0帧相对于位图的坐标：frameX =0，frameY =0;
- 第1帧相对于位图的坐标：frameX = frameW，frameY =0;
- 第2帧相对于位图的坐标：frameX = 2 × frameW，frameY =0;
- ……
- 第6帧相对于位图的坐标：frameX =0，frameY =frameH;
- 第7帧相对于位图的坐标：frameX =frameW，frameY =frameH;
- 第8帧相对于位图的坐标：frameX =2 × frameW，frameY =frameH;
- ……

由此就可以得出每一帧相对于整个位图的坐标公式：

- 下标为N帧的X坐标：frameX=n%col × frameW;
- 下标为N帧的Y坐标：frameY=n/col × frameH;

得到下标为N的这一帧位置后，如果想让这一帧显示在可视区域中，那么这一帧的坐标应该是：

```
frameX=-n%col×frameW;
frameY=-n/col×frameH;
```

掌握了这个公式后，接下来看封装的drawFrame函数：

```
public void drawFrame(int currentFrame, Canvas canvas, Paint paint) {
        int frameW = bmpRobot.getWidth() / 6;
        int frameH = bmpRobot.getHeight() / 2;
        //得到位图的列数
        int col = bmpRobot.getWidth() / frameW;
        //得到当前帧相对于位图的X坐标
        int x = currentFrame % col * frameW;
        //得到当前帧相对于位图的Y坐标
        int y = currentFrame / col * frameH;
        canvas.save();
        //设置一个宽高与机器人每帧相同大小的可视区域
        canvas.clipRect(robot_x, robot_y, robot_x + bmpRobot.getWidth() / 6,
                        robot_y + bmpRobot.getHeight() / 2);
        if (dir == DIR_LEFT) {//如果是向左侧移动
        //镜像操作
        canvas.scale(-1, 1, robot_x - x + bmpRobot.getWidth() / 2, robot_y
                    - y + bmpRobot.getHeight() / 2);
        }
        canvas.drawBitmap(bmpRobot, robot_x - x, robot_y - y, paint);
        canvas.restore();
}
```

这里要注意的是因为机器人位图上只有朝向右的十二帧动作帧，而没有朝向左侧的帧，所以可以对位图进行镜像反转，使其得到朝向左的所有动作帧。至于如何为画布设置可视区域以及对位图的操作，在之前的章节中已经详细介绍，这里不再赘述。

按键事件处理：

```
@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
        if (keyCode == KeyEvent.KEYCODE_DPAD_UP) {
                robot_y -= 5;
        }
        if (keyCode == KeyEvent.KEYCODE_DPAD_DOWN) {
                robot_y += 5;
        }
        if (keyCode == KeyEvent.KEYCODE_DPAD_LEFT) {
                robot_x -= 5;
                dir = DIR_LEFT;
        }
        if (keyCode == KeyEvent.KEYCODE_DPAD_RIGHT) {
                robot_x += 5;
                dir = DIR_RIGHT;
        }
        return super.onKeyDown(keyCode, event);
}
```

四个方向实体按键对应位图的坐标进行增加或者减小，那么需要注意的是，当用户单击方向的左或者右键时，不要忘记改变当前播放机器人的朝向。

逻辑函数：

```
private void logic() {
        currentFrame++;
        if (currentFrame >= 12) {
                currentFrame = 0;
        }
}
```

逻辑仍然是做所有帧数的循环控制，让其动作帧不间断重复播放；项目运行效果如图 4-46 所示。



图 4-46　角色移动效果图

一般在处理游戏主角行走时，不可能让用户每次都重复点击，这样对玩家来说操作起来很累。一直按下一个方向键，能持续移动主角这才是合理的。

其实 View 提供的实体按键监听的函数 onKeyDown 就已经为开发者提供了此功能，当某一方向按键处于一直被按下的状态时，onKeyDown 也将一直被响应。

但是这里存在一个问题，当一直按下一个方向按键进行移动主角行走时，细心一点观察的话可以看到主角的行走会被卡了一下!

其实这个问题的原因是可以解释的，因为 onKeyDown 函数本身只是用于监听按键按下的事件。当按键按下的状态保持了一定毫秒后，系统会默认进入长按键状态，并不停的响应此函数，那么从一次按键的状态转入长按键状态系统肯定需要一个时间来判定，也正是因为这个原因，才会出当长按键时，主角的移动被卡了一下的现象。

对此进行处理，处理的方式如下：

首先定义四个方向按键的变量：

```
private boolean isUp,isDown,isLeft,isRight;
```

四个变量分别表示四个方向按键是否被按下的状态，初始都默认为 false。

在 onKeyDown 函数中对用户按下的方向按键进行处理：

```
public boolean onKeyDown(int keyCode, KeyEvent event) {
      if (keyCode == KeyEvent.KEYCODE_DPAD_UP) {
            isUp = true;
      }
      if (keyCode == KeyEvent.KEYCODE_DPAD_DOWN) {
            isDown = true;
      }
      if (keyCode == KeyEvent.KEYCODE_DPAD_LEFT) {
            isLeft = true;
            dir = DIR_LEFT;
      }
      if (keyCode == KeyEvent.KEYCODE_DPAD_RIGHT) {
            isRight = true;
            dir = DIR_RIGHT;
      }
      return super.onKeyDown(keyCode, event);
}
```

当用户单击方向按键的任意一个按键时，使按键对应的方向变量设置为 true。

然后在逻辑函数中操作主角的移动：

```
private void logic() {
      if (isUp) {
            robot_y -= 5;
      }
      if (isDown) {
            robot_y += 5;
      }
      if (isLeft) {
            robot_x -= 5;
      }
      if (isRight) {
            robot_x += 5;
      }
      ...
}
```

逻辑中一直对定义的四个方向变量进行判定，只要在按键监听函数中有方向按键被按下时，就会改变定义的方向变量为 true，一旦某一方向变量为 true 之后，逻辑中就会对主角进行移动操作。

当然到此还没有处理结束，因为方向变量一直为 true，逻辑中就会对主角一直进行移动操作，所以此时需要按键抬起函数配合完成最后一步：

```
@Override
public boolean onKeyUp(int keyCode, KeyEvent event) {
      if (keyCode == KeyEvent.KEYCODE_DPAD_UP) {
            isUp = false;
      }
      if (keyCode == KeyEvent.KEYCODE_DPAD_DOWN) {
            isDown = false;
      }
      if (keyCode == KeyEvent.KEYCODE_DPAD_LEFT) {
            isLeft = false;
      }
      if (keyCode == KeyEvent.KEYCODE_DPAD_RIGHT) {
            isRight = false;
      }
      return super.onKeyUp(keyCode, event);
}
```

这样处理后不管玩家是短暂的点击实体的方向按键还是长按方向键，只要按键抬起，监听按键抬起的函数 onKeyUp 就会将相应定义的方向变量设置为 false，停止逻辑对主角的移动操作。

最后还要提醒一点，不管是在按键按下的监听函数还是在按键抬起的监听函数中，对于用户点击的按键键值进行判定时，如果是使用 if(){}else if(){}//...的形式进行判断的话，那么当用户同时点击右方向键以及下方向键时，监听函数只能匹配到一个方向的键值；但是如果对每个方向的键值都是使用 if(){}//...if(){}//...这种形式的话，用户同时按下右方向键与下方向键时，在监听函数中则会匹配到两种键值。

当然在逻辑函数中，对于自定义的四个方向变量也要根据需要选择使用 if 的形式；如果就是想控制主角单方向移动，那就使用 if(){}else if(){}//...的形式吧。

4.14 碰撞检测

手机游戏开发中最常用到三种检测碰撞的方式，分别是：矩形碰撞、圆形碰撞和像素碰撞。实际上，与其说是三种碰撞检测方式倒不如说是两种，其原因会在最后介绍像素碰撞时详细阐述。

4.14.1 矩形碰撞

所谓矩形碰撞就是利用两个矩形之间的位置关系来进行判断，如果一个矩形的像素在另外一个矩形之中，或者之上都可以认为这两个矩形发生了碰撞。

如果单纯去考虑哪些情况会判定两个矩形发生碰撞，倒不如反思维考虑两个矩形之间不发生碰撞的几种情况，这样更容易想到。其实两个矩形不发生碰撞的情况就四种，如图 4-47 所示。

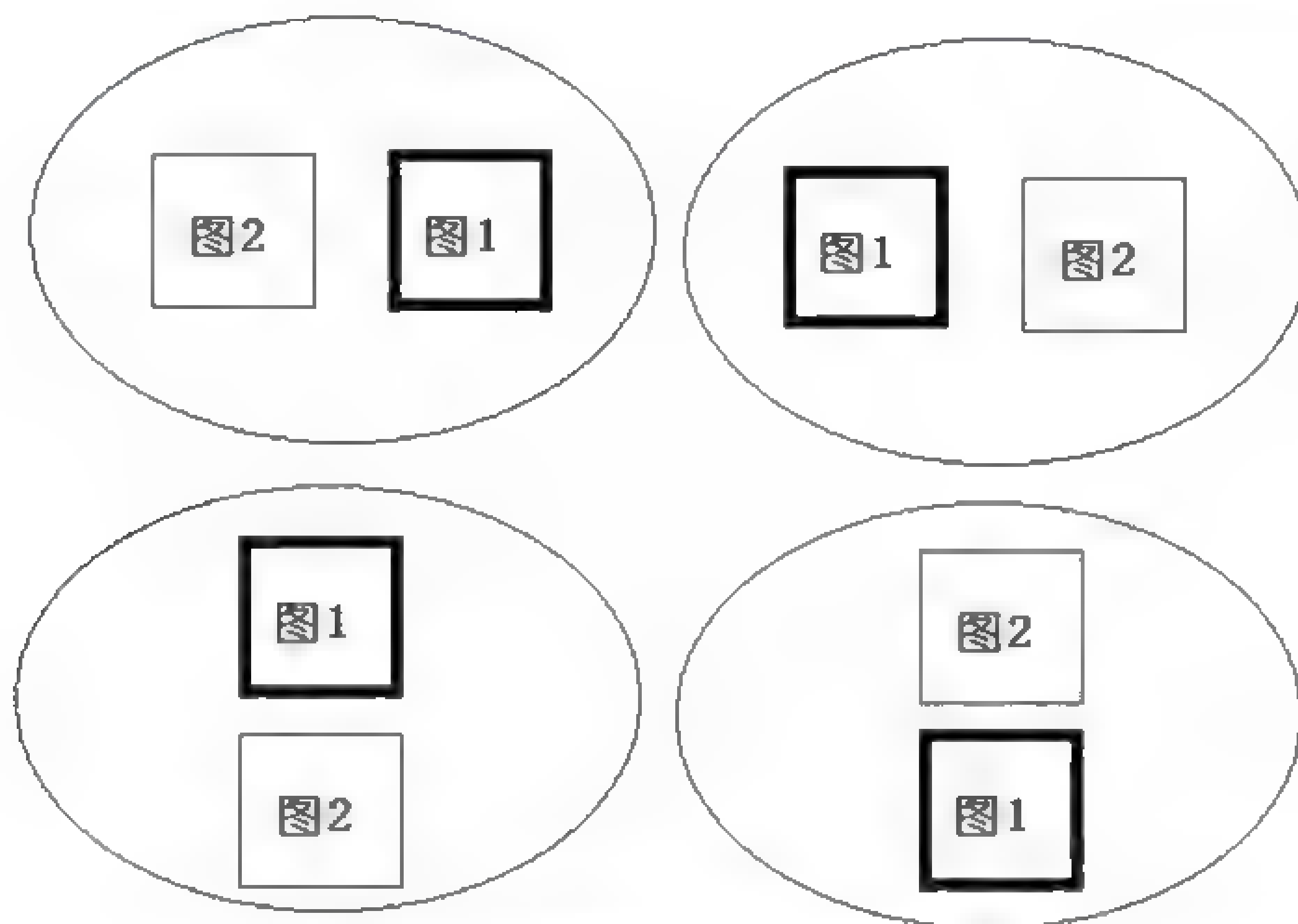


图 4-47 矩形不发生碰撞的四种情况

图 4-47 示意了两个矩形之间永不会发生碰撞的四种情况。下面通过一个实例项目来完成对应的四种判定。

新建项目“RectCollision”，游戏框架为 SurfaceView 游戏框架，项目对应的源代码为“4-14-1（矩形碰撞）”。首先修改 MySurfaceView 类如下：

```
//定义所需的变量:
//定义两个矩形的宽高坐标
private int x1 = 10, y1 = 110, w1 = 40, h1 = 40;
private int x2 = 100, y2 = 110, w2 = 40, h2 = 40;
//便于观察是否发生了碰撞设置一个标识位
private boolean isCollsion;
//然后修改绘图函数:
public  void  myDraw () {
      ...
      //判断是否发生了碰撞
      if (isCollsion) {//发生碰撞
          paint.setColor(Color.RED);
          paint.setTextSize(20);
```

```
            canvas.drawText("Collision! ", 0, 30, paint);
        } else {//没发生碰撞
            paint.setColor(Color.WHITE);
        }
        //绘制两个矩形
        canvas.drawRect(x1, y1, x1 + w1, y1 + h1, paint);
        canvas.drawRect(x2, y2, x2 + w2, y2 + h2, paint);
        ...
}
```

上面代码中，isCollsion 这个变量的存在主要是区分未碰撞和已碰撞。当发生碰撞时（isCollsion 为真），不仅改变了画笔的颜色，还绘制了一句文本信息。绘制文本的原因是因为在书中只显示黑白两色，效果不明显，为了让大家从项目截图中明显的看出其区别而添加的。

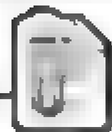
两个矩形默认坐标和宽高值是无法发生碰撞的，所以这里需要对其中一个矩形跟随触屏点进行移动操作，这个操作在触屏事件监听函数中实现：

```
public boolean onTouchEvent(MotionEvent event) {
        //让矩形 1 随着触屏位置移动（触屏点设为此矩形的中心点）
        x1 = (int) event.getX() - w1 / 2;
        y1 = (int) event.getY() - h1 / 2;
        //当矩形之间发生碰撞
        if (isCollsionWithRect(x1, y1, w1, h1, x2, y2, w2, h2)) {
            isCollsion = true;//设置标识位为真
            //当矩形之间没有发生碰撞
        } else {
            isCollsion = false;//设置标识位为假
        }
        return true;
}
```

接下来讲解封装的矩形碰撞的函数：

```
/**
 *
 * @param x1 第一个矩形的 X 坐标
 * @param y1 第一个矩形的 Y 坐标
 * @param w1 第一个矩形的宽
 * @param h1 第一个矩形的高
 * @param x2 第二个矩形的 X 坐标
 * @param y2 第二个矩形的 Y 坐标
 * @param w2 第二个矩形的宽
 * @param h2 第二个矩形的高
 * @return
 */
```

```
public boolean isCollsionWithRect(int x1, int y1, int w1, int h1, int x2,
int y2, int w2, int h2) {
       //当矩形1位于矩形2的左侧
       if (x1 >= x2 && x1 >= x2 + w2) {
              return false;
       //当矩形1位于矩形2的右侧
       } else if (x1 <= x2 && x1 + w1 <= x2) {
              return false;
       //当矩形1位于矩形2的上方
       } else if (y1 >= y2 && y1 >= y2 + h2) {
              return false;
       //当矩形1位于矩形2的下方
       } else if (y1 <= y2 && y1 + h1 <= y2) {
              return false;
       }
       //所有不会发生碰撞都不满足时，肯定就是碰撞了
       return true;
}
```



注意

在两个矩形之间进行碰撞检测时，不仅仅要判定两者 X、Y 坐标之间的位置关系，还要考虑到两个矩形的宽度与高度。

项目运行效果如图 4-48 所示。



图 4-48　矩形碰撞效果图

4.14.2　圆形碰撞

圆形之间的碰撞，主要是利用两圆心的圆心距进行判定的：当两圆的圆心距小于两圆半

径之和时，判定发生了碰撞。下面用一个范例进行说明。

新建项目“CircleCollision”，游戏框架为 SurfaceView 游戏框架，项目对应的源代码为“4-14-2（圆形碰撞）”。这里主要分析一下圆形碰撞的检测方法，其余代码可以自行查看源代码。

将圆形碰撞函数封装为一个方法 isCollisionWithCircle：

```
/**
 * 圆形碰撞
 * @param x1 圆形 1 的圆心 X 坐标
 * @param y1 圆形 2 的圆心 X 坐标
 * @param x2 圆形 1 的圆心 Y 坐标
 * @param y2 圆形 2 的圆心 Y 坐标
 * @param r1 圆形 1 的半径
 * @param r2 圆形 2 的半径
 * @return
 */
private boolean isCollisionWithCircle(int x1, int y1, int x2, int y2,
int r1, int r2) {
      //Math.sqrt:开平方
      //Math.pow(double x, double y): X 的 Y 次方
      if (Math.sqrt(Math.pow(x1 - x2, 2) + Math.pow(y1 - y2, 2))
                                   <= r1 + r2) {
            //如果两圆的圆心距小于或等于两圆半径则认为发生碰撞
            return true;
      }
      return false;
}
```

项目运行效果如图 4-49 所示。



图 4-49　圆形碰撞

4.14.3 像素碰撞

对于碰撞检测已经介绍了矩形与圆形两种方式，其实使用这两种检测方式不是很精确。比如两张大小相同的带透明度的png图，如图4-50所示。



图4-50 示意图1

每张位图的外侧矩形表示每张位图的大小边界，每张位图中间填充的黑色圆形表示有像素的点，每张位图中空白（白色）区域则表示png位图中的透明像素。这种带透明像素的图形之间，如果利用矩形来进行碰撞，肯定不能以图的大小进行碰撞检测！

以两张位图大小来进行碰撞检测的示意图如图4-51所示。

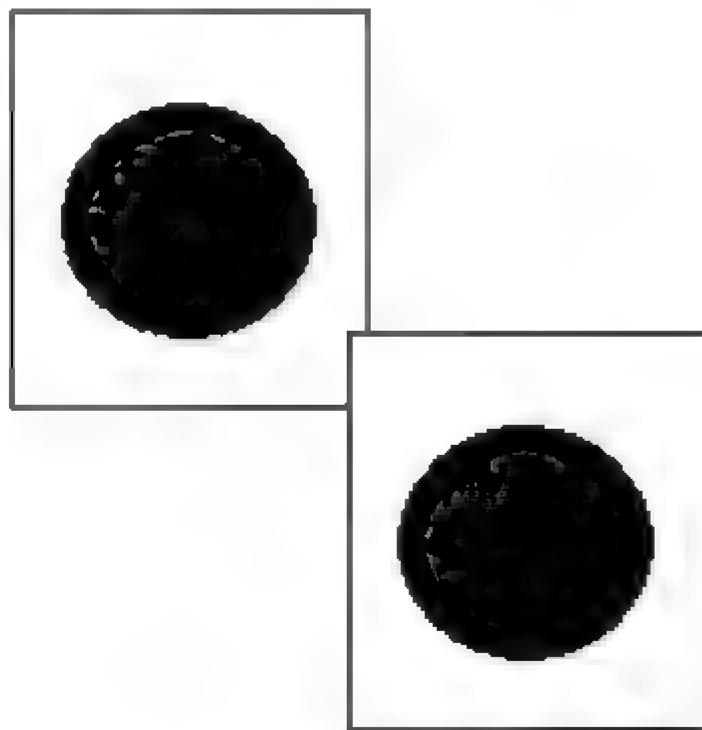


图4-51 示意图2

按照两张位图大小进行检测碰撞，那么如图4-51所示的就是两张位图已经发生了碰撞的情况。但是大家思考一下，如果将图4-51所示的这样两张带透明像素的图绘制在画布上，并且两个位图的位置关系也如图4-51所示，站在玩家的角度来说，他们关注的只有两张位图上非透明像素的两个圆形，很明显这两个圆形并没有发生碰撞，对于玩家来说这是无法接受的！虽然事实上两张位图确实发生了碰撞，但是对玩家而言，这是个Bug。

出现此类问题的原因就是碰撞区域的不精确！当然为了让其碰撞更加的真实，解决的办法也有很多：

- 第一种方法是仍然利用矩形碰撞的方法，但是要设置矩形碰撞区域大小，其大小最好

是刚刚好包住位图的黑色圆形（非透明像素区域）；

- 第二种方法就是直接利用圆形碰撞，而碰撞的圆形区域与位图的黑色圆形（非透明像素）大小一致。

针对示意图表示的这样两张 png 位图来说，虽然圆形碰撞方法明显会优胜于矩形碰撞，但是一旦需要检测碰撞的两张 png 带透明像素的图，并且两张图的非透明像素区域又不是规则的图形，那又该如何更真实的模拟碰撞呢？此时就体现出了像素碰撞的优势！

像素碰撞是怎么模拟碰撞的呢？首先遍历算出一张位图所有的像素点坐标，然后与另外一张位图上的所有点坐标进行对比，一旦有一个像素点的坐标相同，就立刻取出这两个坐标相同的像素点，通过位运算取出这两个像素点的最高位（透明度）进行对比，如果两个像素点都是非透明像素则判定这两张位图发生碰撞。

介绍了像素碰撞之后可以得到两个结论：

- 像素碰撞很精确，不论位图之间是否带有透明像素，都可以精确判断；
- 正是因为像素碰撞的这种高精确判定，从而也会造成代码效率明显降低！
  - ◆ 假设两张 100 × 100 大小的位图利用像素级检测碰撞，仅是遍历两张位图的像素点就要循环 100 × 100 × 2=20000 句逻辑代码；况且还要对筛选出来的相同坐标的像素点进行遍历对比其透明值！这种效率可想而知！

当然，这里的像素碰撞只是大致提供一种思路，肯定还可以进行代码优化；但是不论再优的代码，使用像素级进行碰撞检测终会导致整个程序的运行效率大大降低。因此像素级别的碰撞检测在手机游戏开发中是尽量避免使用的！所以这里也不再详细讲解。

像素级的碰撞检测是不推荐使用的，但是它的精确程度是其他方法无法代替的；不过，这并不代表游戏开发中就没有更精确的检测方式了！

一般游戏开发中，取代像素级碰撞检测的方法是利用“多矩形”、“多圆形”的检测方式来实现的。

4.14.4 多矩形碰撞

所谓多矩形碰撞，顾名思义就是设置多个矩形碰撞区域，如图 4-52 所示。



图 4-52 不规则图例

图 4-52 的左侧是原图，右侧则是在原图的基础上设置了两个矩形碰撞区域。观察图中右

侧可明显看出两个矩形正好将原图中非透明的像素点包起来。这样做的好处是：

- 更精确的检测碰撞方法，精确度基本同于像素级；
- 相对于像素级的碰撞检测，这种做法效率更高。

新建项目“MoreRectCollision”，游戏框架为 SurfaceView 游戏框架，项目对应的源代码为“4-14-4（多矩形碰撞）”。

对多个矩形进行碰撞检测，首先应该处理一个矩形的碰撞检测。关于矩形的碰撞检测，在之前介绍矩形碰撞时已经封装过方法，这里对其代码进行修改：

```
public boolean isCollsionWithRect(Rect rect, Rect rect2) {
        //x1, y1：矩形 1 的左上角
        int x1 = rect.left;
        int y1 = rect.top;
        //w1:矩形 1 的宽
        int w1 = rect.right - rect.left;
        //h1:矩形 1 的高
        int h1 = rect.bottom - rect.top;
        //x2, y2：矩形 2 的左上角
        int x2 = rect2.left;
        int y2 = rect2.top;
        //w2:矩形 2 的宽
        int w2 = rect2.right - rect2.left;
        //h2:矩形 2 的高
        int h2 = rect2.bottom - rect2.top;
        if (x1 >= x2 && x1 >= x2 + w2) {
              return false;
        } else if (x1 <= x2 && x1 + w1 <= x2) {
              return false;
        } else if (y1 >= y2 && y1 >= y2 + h2) {
              return false;
        } else if (y1 <= y2 && y1 + h1 <= y2) {
              return false;
        }
        return true;
}
```

Rect 类中定义了矩形坐标属性 top、bottom、left、right。

- left：表示矩形左上角坐标的 X 坐标
- top：表示矩形左上角坐标的 Y 坐标
- right：表示矩形右下角的 X 坐标
- bottom：表示矩形右下角的 Y 坐标

对一个矩形的碰撞检测封装好了，下面就再次对其进行修改，使之支持多矩形碰撞

检测：

```
public boolean isCollsionWithRect(Rect[] rectArray, Rect[] rect2Array) {
      Rect rect = null;
      Rect rect2 = null;
      for (int i = 0; i < rectArray.length; i++) {
             //依次取出第一个矩形数组的每个矩形实例
             rect = rectArray[i];
             //获取到第一个矩形数组中每个矩形元素的属性值
             int x1 = rect.left + this.rectX1;
             int y1 = rect.top + this.rectY1;
             int w1 = rect.right - rect.left;
             int h1 = rect.bottom - rect.top;
             for (int j = 0; j < rect2Array.length; j++) {
                    //依次取出第二个矩形数组的每个矩形实例
                    rect2 = rect2Array[j];
                    //获取到第二个矩形数组中每个矩形元素的属性值
                    int x2 = rect2.left + this.rectX2;
                    int y2 = rect2.top + this.rectY2;
                    int w2 = rect2.right - rect2.left;
                    int h2 = rect2.bottom - rect2.top;
                    //进行循环遍历两个矩形碰撞数组所有元素之间的位置关系
                    if (x1 >= x2 && x1 >= x2 + w2) {
                    } else if (x1 <= x2 && x1 + w1 <= x2) {
                    } else if (y1 >= y2 && y1 >= y2 + h2) {
                    } else if (y1 <= y2 && y1 + h1 <= y2) {
                    } else {
                      //只要有一个碰撞矩形数组与另一碰撞矩形数组发生碰撞则认为碰撞
                      return true;
                    }
             }
      }
      return false;
}
```

上面代码就是遍历两个碰撞矩形数组每个矩形之间的位置关系，一旦有一个矩形数组中的矩形与另外一个矩形数组的矩形发生碰撞就可认为发生了多矩形碰撞。

项目运行效果如图 4-53 所示。

由于多圆形的碰撞检测类似于多矩形碰撞，所以这里就不再赘述。

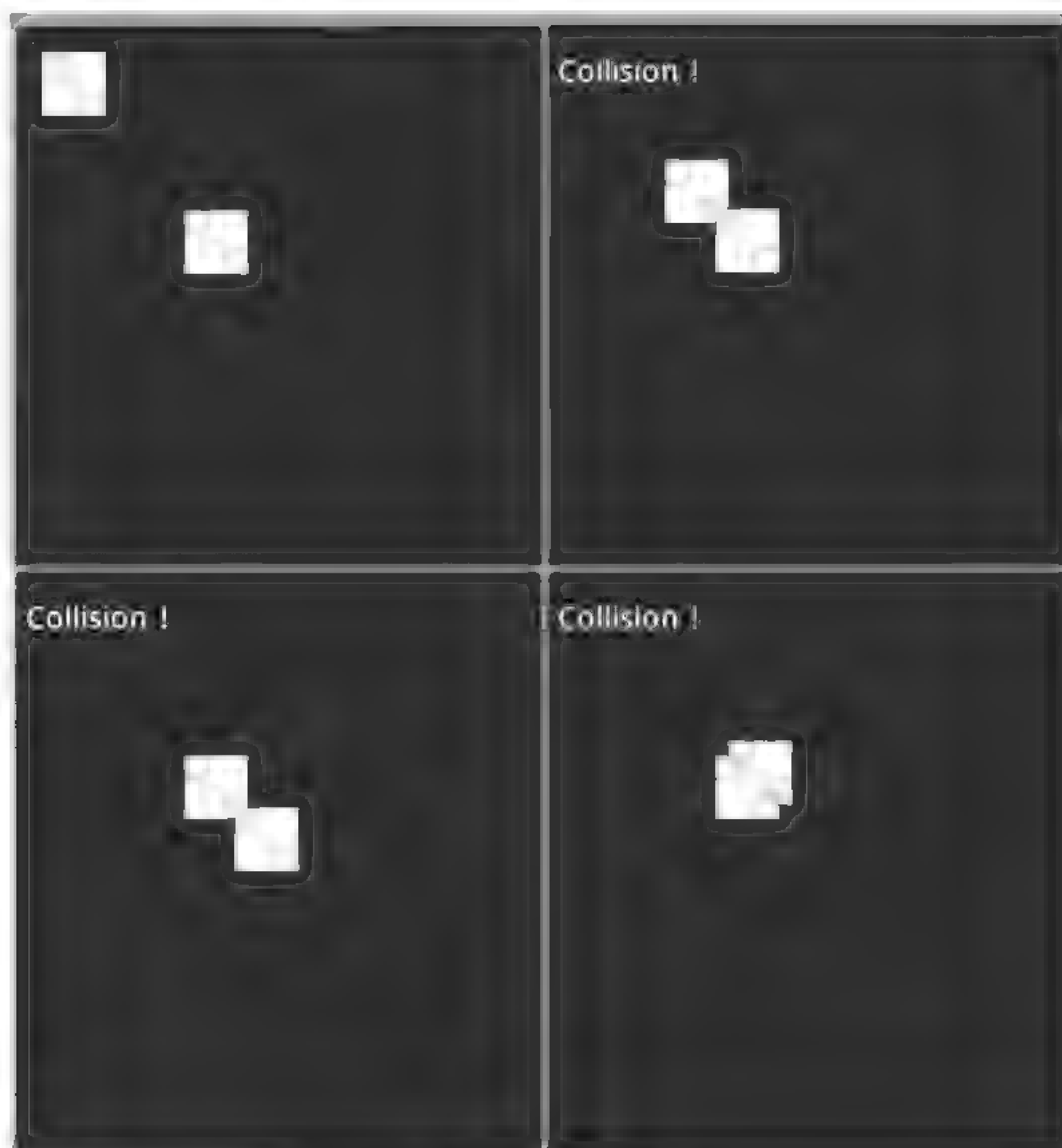


图 4-53　多矩形碰撞

4.14.5　Region 碰撞检测

在之前介绍过 Region 这个类，其实此类还有一个比较常用的方法就是用于判断一个点是否在矩形区域内，其方法是使用 Region 类中的 contains（int x, int y）函数。

M Contains（int x, int y）

作用：用于判断一个点是否在 Region 的矩形区域中

两个参数：点的 X、Y 坐标

新建项目“RegionCollision”，游戏框架为 SurfaceView 游戏框架，项目对应的源代码为“4-14-5（Region 碰撞检测）”。修改 MySurfaceView 类代码如下：

```
//定义碰撞矩形
private Rect rect = new Rect(0, 0, 50, 50);
//定义 Region 类实例
private Region r = new Region(rect);
//表示是否发生碰撞的标识位
private boolean isInclude;
```

```
//绘图函数:
public  void  myDraw () {
        ...
        //标识位为真时，绘制 icon 图
        if (isInclude) {
                    canvas.drawBitmap(BitmapFactory.decodeResource(this
                          .getResources(), R.drawable.icon), 100, 50, paint);
        }
        //绘制矩形区域（便于观察）
        canvas.drawRect(rect, paint);
        ...
}

//触屏事件:
public boolean onTouchEvent(MotionEvent event) {
        //判定用户触屏的坐标点是否在碰撞矩形内
        if (r.contains((int) event.getX(), (int) event.getY())) {
              isInclude = true;
        } else {
              isInclude = false;
        }
        return true;
}
```

项目运行效果如图 4-54 所示。图中的左侧是运行初始效果，右侧是当用户触屏位置在白色碰撞矩形内的效果。



图 4-54　Region 碰撞检测

4.15 游戏音乐与音效

在一款游戏中，除了华丽的界面 UI 直接吸引玩家外，另外重要的就是游戏的背景音乐与音效；合适的背景音乐以及精彩的音效搭配会令整个游戏上升一个档次。

在 Android 中，常用于播放游戏背景音乐的类是 MediaPlayer，而用于游戏音效的则是 SoundPool 类，至于 MediaPlayer 与 SoundPool 之间的区别将在讲解两者的使用方法后详细的进行分析。

4.15.1 MediaPlayer

MediaPlayer 实例化不是 new 出来的，而是通过调用静态方法 create（Context context, int resid）得到的。除获取实例以外，MediaPlayer 类常用的函数如下：

- prepare(): 为播放音乐文件做准备工作。
- start(): 播放音乐。
- pause(): 暂停音乐播放。
- stop(): 停止音乐播放。

暂停音乐与停止音乐，主要的区别在于：暂停音乐播放后，可继续播放，再次调用 start()函数即可；停止音乐播放后，无法继续播放，必须重新做播放音乐的准备工作 prepare()，然后再调用 start()函数进行播放音乐。

Mediaplayer 类的其他常用函数：

1. setLooping(boolean looping)

作用：设置音乐是否循环播放

参数：true 表示循环播放，false 表示不循环播放

2. seekTo(int msec)

作用：将音乐播放跳转到某一时间点

参数：跳转时间(以毫秒为单位)

3. getDuration()

作用：获取播放的音乐文件总时间长度

4. getCurrentPosition()

作用:得到当前播放音乐的时间点

除此之外，还需介绍音乐管理类 AudioManager，它提供了获取当前音乐大小以及最大音量等。

AudioManager 类常用函数：

1. setStreamVolume(int streamType, int index, int flags)

作用：设置音量大小

第一个参数：音量类型（音乐的常量：AudioManager.STREAM_MUSIC）

第二个参数：音量大小

第三个参数：设置一个或者多个标识

2. getStreamVolume(int streamType)

作用：获取当前音量大小

参数：获取音量大小的类型

3. getStreamMaxVolume(int streamType)

作用：获取当前音量最大值

参数：获取音量大小的类型

Android OS 中，如果去按手机上调节音量的按钮，会遇到两种情况，一种是调整手机本身的铃声音量，另外一种是调整游戏、软件的音乐播放的音量。

在游戏中的时候，默认调整的是手机的铃声音量，只有游戏中有声音在播放的时候，才能去调整游戏的音量。因此往游戏中添加音乐时，需要使用如下函数：

M Activity.setVolumeControlStream(int streamType)

作用：设置控制音量的类型

参数：音量类型(AudioManager.STREAM_MUSIC：媒体音量)

在熟习了这些类以及每个类常用的函数后，就实战编写一个简略的播放器，可实现快进、快退、播放、暂停、调整音量大小的操作。

新建项目“MediaPlayerProject”，游戏框架为 SurfaceView 游戏框架，项目对应的源代码为“4-15-1（MediaPlayer 音乐）”。首先看看项目效果，如图 4-55 所示。



图 4-55　MediaPlayer 项目效果图

修改 MySurfaceView 类：

```
//声明音乐的状态常量
private final int MEDIAPLAYER_PAUSE = 0;//暂停
private final int MEDIAPLAYER_PLAY = 1;//播放中
private final int MEDIAPLAYER_STOP = 2;//停止
//音乐的当前的状态
```

```
private int mediaSate = 0;
//声明一个音乐播放器
private MediaPlayer mediaPlayer;
//当前音乐播放的时间点
private int currentTime;
//当前音乐的总时间
private int musicMaxTime;
//当前音乐的音量大小
private int currentVol;
//快进、快退时间戳
private int setTime = 5000;
//播放器管理类
private AudioManager am;
```

视图初始化：

```
public void surfaceCreated(SurfaceHolder holder) {
      ...
      //实例音乐播放器
      mediaPlayer = MediaPlayer.create(context, R.raw.bgmusic);
      //设置循环播放
      mediaPlayer.setLooping(true);//设置循环播放
      //获取音乐文件的总时间
      musicMaxTime = mediaPlayer.getDuration();
      //实例管理类
      am=(AudioManager)MainActivity.instance.getSystemService(Co
                                          ntext.AUDIO_SERVICE);
      //设置当前调整音量大小只是针对媒体音乐进行调整
      MainActivity.instance.setVolumeControlStream(AudioManager.
                                            STREAM_MUSIC);
      ...
}
```

绘图函数：

```
public  void  myDraw () {
      ...
      canvas.drawColor(Color.WHITE);
      paint.setColor(Color.RED);
      paint.setTextSize(15);
      canvas.drawText("当前音量: " + currentVol, 50, 40, paint);
      canvas.drawText("当前播放的时间(毫秒)/总时间(毫秒)", 50, 70,
                        paint);
      canvas.drawText(currentTime + "/" + musicMaxTime, 100, 100, paint);
      canvas.drawText("方向键中间按钮切换 暂停/开始", 50, 130,
paint);
      canvas.drawText("方向键←键快退" + setTime / 1000 + "秒 ", 50,
```

```
160, paint);
        canvas.drawText("方向键→键快进" + setTime / 1000 + "秒 ", 50,
190, paint);
        canvas.drawText("方向键↑键增加音量 ", 50, 220, paint);
        canvas.drawText("方向键↓键减小音量", 50, 250, paint);
        ...
}
```

因为音乐播放的当前音量、当前时间是随时发生变化的，所以应该在逻辑函数中不断的得到，以确保是最新值。

逻辑函数：

```
private void logic() {
        if (mediaPlayer != null) {
                //获取当前音乐播放的时间
                currentTime = mediaPlayer.getCurrentPosition();
                //获取当前的音量值
                currentVol = am.getStreamVolume(AudioManager.STREAM_MUSIC);
        } else {
                currentTime = 0;
        }
}
```

实体按键监听函数：

```
public boolean onKeyDown(int keyCode, KeyEvent event) {
        //导航中键播放/暂停操作
        if (keyCode == KeyEvent.KEYCODE_DPAD_CENTER) {
                try {
                        switch (mediaSate) {
                        //当前处于播放的状态
                        case MEDIAPLAYER_PLAY:
                                mediaPlayer.pause();
                                mediaSate = MEDIAPLAYER_PAUSE;
                                break;
                        //当前处于暂停的状态
                        case MEDIAPLAYER_PAUSE:
                                mediaPlayer.start();
                                mediaSate = MEDIAPLAYER_PLAY;
                                break;
                        //当前处于停止的状态
                        case MEDIAPLAYER_STOP:
                                if (mediaPlayer != null) {
                                        mediaPlayer.pause();
                                        mediaPlayer.stop();
                                }
```

```
                    mediaPlayer.prepare();
                    mediaPlayer.start();
                    mediaSate = MEDIAPLAYER_PLAY;
                    break;
                }
            } catch (IllegalStateException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        //导航上键调整音乐播放声音变大
        } else if (keyCode == KeyEvent.KEYCODE_DPAD_UP) {
            am.setStreamVolume(AudioManager.STREAM_MUSIC, currentVol + 1,
AudioManager.FLAG_PLAY_SOUND);
        //导航下键调整音乐播放声音变小
        } else if (keyCode == KeyEvent.KEYCODE_DPAD_DOWN) {
            am.setStreamVolume(AudioManager.STREAM_MUSIC, currentVol - 1,
AudioManager.FLAG_PLAY_SOUND);
        //导航左键调整音乐播放时间倒退五秒
        } else if (keyCode == KeyEvent.KEYCODE_DPAD_LEFT) {
            if (currentTime - setTime <= 0) {
                mediaPlayer.seekTo(0);
            } else {
                mediaPlayer.seekTo(currentTime - setTime);
            }
        //导航右键调整音乐播放时间快进五秒
        } else if (keyCode == KeyEvent.KEYCODE_DPAD_RIGHT) {
            if (currentTime + setTime >= musicMaxTime) {
                mediaPlayer.seekTo(musicMaxTime);
            } else {
                mediaPlayer.seekTo(currentTime + setTime);
            }
        }
        return super.onKeyDown(keyCode, event);
}
```

MediaPlayer 的使用很简单。创建一个实例，然后调用 prepare()准备函数，之后就可以播放音乐了。除了对 MediaPlayer 常用的操作外，Android 还提供了一个接口：

```
MediaPlayer.OnCompletionListener
```

其作用是监听音乐是否完全播放完毕。使用这个接口需要注意两点：

①必须重写一个抽象函数：

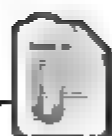
M onCompletion（MediaPlayer arg0）

作用：音乐播放完毕会响应此函数

参数：完成音乐播放的 MediaPlayer 实例

②将需要的 MediaPlayer 实例绑定在完成监听器上：

setOnCompletionListener（OnCompletionListener listener）



注意

这个监听音乐播放是否完成的监听器，只能针对音乐只播放一次的情况进行监听。也就是说，如果设置了音乐循环播放，那么监听器永远都不会监听到音乐是否播放完成！

4.15.2 SoundPool

除了 MediaPlayer 能播放音乐外，SoundPool 也能播放一些音乐文件，它们之间最大的区别是 SoundPool 只能播放小的文件，至于更详细的区别后文再进行讲解。

SoundPool 类的构造函数如下：

C SoundPool（int maxStreams, int streamType, int srcQuality）

作用：实例化一个 SoundPool 实例

第一个参数：允许同时播放的声音最大值

第二个参数：声音类型

第三个参数：声音的品质

SoundPool 类中常用的函数如下：

M int load（Context context, int resId, int priority）

作用：加载音乐文件，返回音乐 ID（音乐流文件数据）

第一个参数：Context 实例

第二个参数：音乐文件 Id

第三个参数：标识优先考虑的声音。目前使用没有任何效果，只是具备了兼容性价值

M int play（int soundID, float leftVolume, float rightVolume, int priority, int loop, float rate）

作用：音乐播放，播放失败返回 0，正常播放返回非 0 值

第一个参数：加载后得到的音乐文件 ID

第二个参数：音量的左声道，范围：0.0~1.0

第三个参数：音量的右声道，范围：0.0~1.0

第四个参数：音乐流的优先级，0 是最低优先级

第五个参数：音乐的播放次数，-1 表示无限循环，0 表示正常一次，大于 0 则表示循环次数

第六个参数：播放速率，取值范围：0.5~2.0，1.0 表示正常播放

M pause（int streamID）

作用：暂停音乐播放

参数:音乐文件加载后的流 ID

M stop（int streamID）

作用：结束音乐播放

参数: 音乐文件加载后的流 ID

- release()

作用：释放 SoundPool 的资源

- setLoop（int streamID, int loop）

作用：设置循环次数

第一个参数：音乐文件加载后的流 ID

第二个参数：循环次数

- setRate（int streamID, float rate）

作用：设置播放速率

第一个参数：音乐文件加载后的流 ID

第二个参数：速率值

- setVolume（int streamID, float leftVolume, float rightVolume）

作用：设置音量大小

第一个参数：音乐文件加载后的流 ID

第二个参数：左声道音量

第三个参数：右声道音量

- setPriority（int streamID, int priority）

作用：设置流的优先级

第一个参数：音乐文件加载后的流 ID

第二个参数：优先级值

下面就通过播放音乐文件的实例来详细讲解如何使用 SoundPool。新建项目“SoundPoolProject”，游戏框架为 SurfaceView 游戏框架，项目对应的源代码为“4-15-2（SoundPool 音效）”。首先导入音乐文件，如图 4-56 所示。



图 4-56　声音文件

两个音乐文件分别为：himi_long.mid，音乐长度 42 秒；himi_short.ogg，音乐长度 1 秒。

修改 MySurfaceVIew 类：

```
//声明 SoundPool
private SoundPool sp;
//记录长音乐文件 id
private int soundId_long;
//记录断短音乐文件 id
private int soundId_short;
```

构造函数：

```
public MySurfaceView(Context context) {
      ...
      //实例 SoundPool 播放器
      sp = new SoundPool(4, AudioManager.STREAM_MUSIC, 100);
      //加载音乐文件获取其数据 ID
      soundId_long = sp.load(context, R.raw.himi_long, 1);
      //加载音乐文件获取其数据 ID
      soundId_short = sp.load(context, R.raw.himi_short, 1);
      ...
}
```

绘图函数：

```
public void myDraw() {
      ...
      paint.setColor(Color.RED);
      paint.setTextSize(15);
      canvas.drawText("点击导航键的上键：播放断音效", 50, 50, paint);
      canvas.drawText("点击导航键的下键：播放长音效", 50, 80, paint);
      ...
}
```

实体按键监听函数：

```
public boolean onKeyDown(int keyCode, KeyEvent event) {
      if (keyCode == KeyEvent.KEYCODE_DPAD_UP)
            sp.play(soundId_long, 1f, 1f, 0, 0, 1);
      else if (keyCode == KeyEvent.KEYCODE_DPAD_DOWN)
            sp.play(soundId_short, 2, 2, 0, 0, 1);
      return super.onKeyDown(keyCode, event);
}
```

项目效果如图 4-57 所示。



图 4-57　SoundPool 项目截图

整个项目的流程很简单，通过判断用户的按键，播放不同的音乐文件；但是运行项目时会报出如图 4-58 所示的错误：

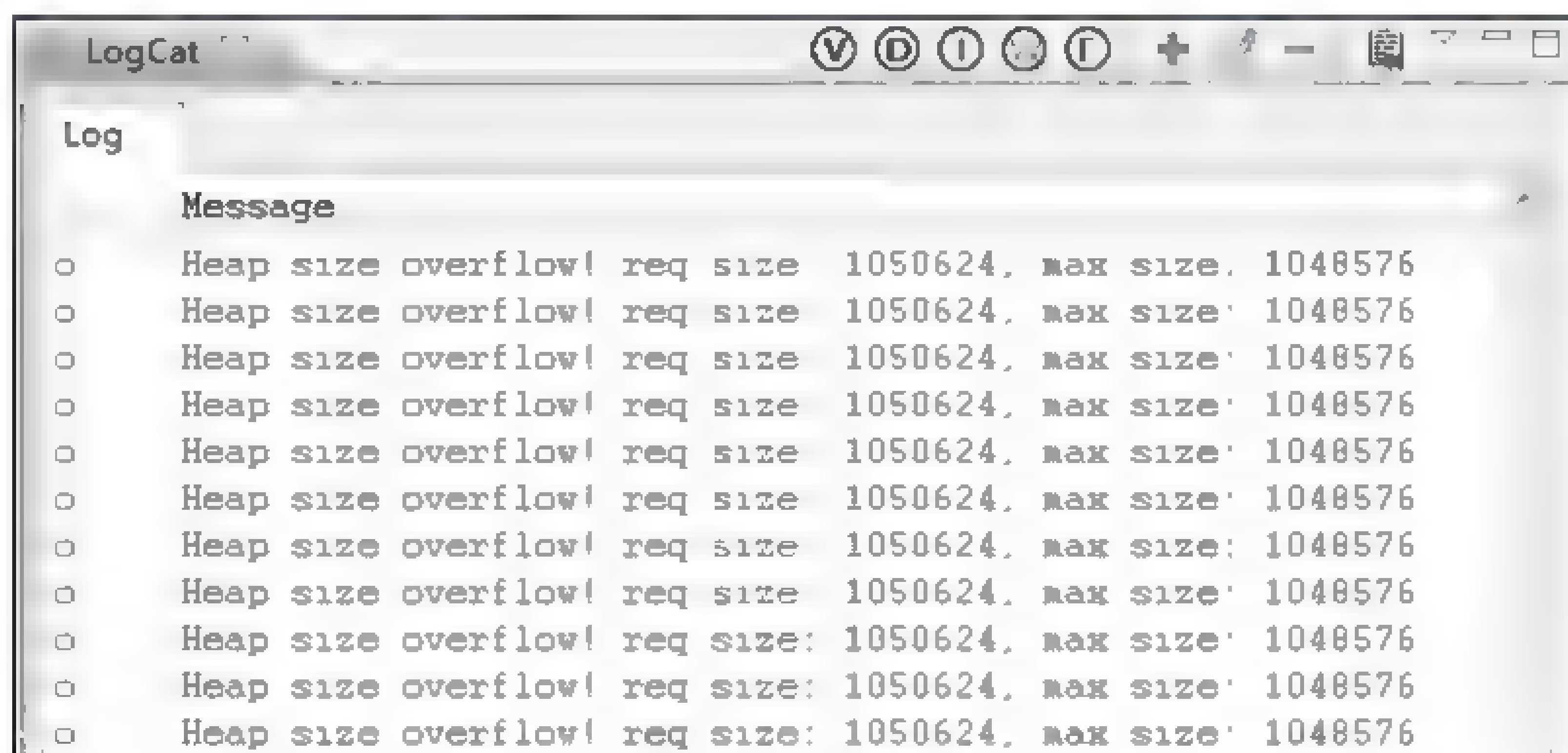


图 4-58　异常截图

错误对应的程序代码是加载长音乐文件生成其数据 ID 一行，出现此错误的原因如下：

利用 SoundPool 播放音乐文件，首先都会对需要播放的音乐文件通过函数 int load（Context context, int resId, int priority）进行加载，并且生成对应的音乐数据 ID；其生成的数据 ID（int 值）就是整个音乐文件的所有数据，而当前项目的长音乐文件 himi_long.mid，音乐长度有 42 秒，其中的音乐流数据文件也远远超过了 int 的最大值，所以当程序加载此音乐文件生成对应的数据 ID 时，会报超过最大值的异常。

虽然出现此异常，但是还不会导致整个程序崩溃，只是当再播放长的音乐文件时，会发现播放的时间很短，明显的感觉到像被剪切了一样；这也证实了 SoundPool 只能存放 1M 大小的音乐数据。

4.15.3 MediaPlayer 与 SoundPool 优劣分析

1. 使用 MediaPlayer 的优缺点

（1）缺点

资源占用量较高、延迟时间较长、不支持多个音频同时播放等。

除此之外使用 MediaPlayer 进行播放音乐时，尤其是在快速连续播放声音（比如连续猛点按钮）时，会非常明显的出现 1~3 秒左右的延迟；当然此问题可以使用 MediaPlayer.seekTo() 这个方法来解决。

（2）优点

支持很大的音乐文件播放，而且不会同 SoundPool 一样需要加载准备一段时间，MediaPlayer 能及时播放音乐。

2. 使用 SoundPool 的优缺点

（1）缺点

①最大只能申请 1M 的内存空间，这就意味着用户只能使用一些很短的声音片段，而不能用它来播放歌曲或者游戏背景音乐。

②SoundPool 提供了 pause 和 stop 方法，但建议最好不要轻易使用这些方法，因为使用它们可能会导致程序莫名其妙的终止。

③使用 SoundPool 时音频格式建议使用 OGG 格式。如果使用 WAV 格式的音频文件，在播放的情况下有时会出现异常关闭的情况。

④在使用 SoundPool 播放音乐文件的时候，如果在构造中就调用播放函数进行播放音乐，其效果则是没有声音！不是因为函数没有执行，而是 SoundPool 需要加载准备时间！当然这个准备时间也很短，不会影响使用，只是程序一运行播放刚开始会没有声音罢了。

（2）优点

支持多个音乐文件同时播放。

通过以上的分析可以明显的知道，在 Android 游戏开发中，游戏背景音乐使用 MediaPlayer 肯定比使用 SoundPool 要合适；而游戏音效的播放采用 SoundPool 则更好，毕竟游戏中肯定会出现多个音效同时进行播放的情况。

4.16 游戏数据存储

对于数据的存储，Android 提供了 4 种保存方式。

（1）SharedPreference

此方式适用于简单数据的保存，文如其名，属于配置性质的保存，不适合数据比较大的

情况，默认存放在手机内存里。

（2）FIleInputStream/FileOutputStream

此方式比较适合游戏的保存和使用，流文件数据存储可以保存较大的数据，而且通过此方式不仅能把数据存储在手机内存中，也能将数据保存到手机的SDcard中。

（3）SQLite

此方式也适合游戏的保存和使用，不仅可以保存较大的数据，而且可以将自己的数据存储到文件系统或者数据库当中，如SQLite数据库，也能将数据保存到SDcard中。

（4）ContentProvider

此方式不推荐用于游戏保存，虽然此方式能存储较大数据，还支持多个程序之间的数据进行交换，但由于游戏中基本就不可能去访问外部应用的数据，所以对于此方式在本书中就不予讲解，有兴趣的可以自行查阅相关书籍和资料。

4.16.1 SharedPreference

SharedPreference实例是通过Context对象得到的：

M Context.getSharedPreferences（String name, int mode）

作用：利用Context对象获取一个SharedPreference实例

第一个参数：生成保存记录的文件名

第二个参数：操作模式

SharedPreference实例的操作模式一共有四种：

- Context.MODE_PRIVATE：新内容覆盖原内容。
- Context.MODE_APPEND：新内容追加到原内容后。
- Context.MODE_WORLD_READABLE：允许其他应用程序读取。
- Context.MODE_WORLD_WRITEABLE：允许其他应用程序写入，会覆盖原数据。

SharedPreference常用函数：

- getFloat（String key, float defValue）
- getInt（String key, int defValue）
- getLong（String key, long defValue）
- getString（String key, String defValue）
- getBoolean（String key, boolean defValue）

SharedPreference常用函数的作用是获取存储文件中的值，根据方法不同获取不同对应的类型值，一般第一个参数为索引Key值，第二个参数为在存储文件中找不到对应Value值时，默认的返回值。其实SharedPreference的存储数据和读取的方式都类似哈希表，这里第二个参数需要传入一个默认返回值，这也避免了找不到对应Key的Value值时，出现返回异常。

以上是对存储文件中进行读取数据的一些常用操作函数，当然对应的肯定也有保存数据时的这些类型函数。但是在对存储文件的数据进行存入操作时，首先需要利用SharedPreference实例得到一个编辑对象：

```
SharedPreferences.Editor edit ();
```

得到编辑对象之后就可以对SharedPreference中的数据进行操作。

- SharedPreferences.Editor.putFloat（arg0, arg1）
- SharedPreferences.Editor.putInt（arg0, arg1）
- SharedPreferences.Editor.putLong（arg0, arg1）
- SharedPreferences.Editor.putString（arg0, arg1）
- SharedPreferences.Editor.putBoolean（arg0, arg1）

以上方法的作用是对存储的数据进行操作（写入、保存），其中的第一个参数是需要保存数据的Key值索引，第二个参数是需要保存的数据。

到此虽然对SharedPreferences中的数据进行了修改或者保存，但是还没有真正的写入到SharedPreferences生成的存储文件中，所以还需要将其编辑的数据进行提交方可完成存入和修改：

```
SharedPreferences.Editor.commit()
```

当提交之后，整个保存的步骤才真正的结束，在使用SharedPreferences进行存储数据时，务必不能忘记最后一步的提交！

除此之外，如果想删除存储文件中的一条数据，则可以使用以下函数：

```
SharedPreferences.Editor.clear()
```

为了让大家熟习在游戏开发中如何嵌入游戏存储，下面简单写一个小游戏，然后通过SharedPreference为游戏添加保存功能。小游戏很简单，绘制9个方格，初始在第一格子有个圆形，用户可以通过手机实体的左右导航键移动圆形，上键代表保存游戏状态，下键代表读取游戏状态。

新建项目“SharedPreferenceProject”，游戏框架为SurfaceView游戏框架，项目对应的源代码为“4-16-1（游戏保存之SharedPreference）”。

修改MySurfaceView如下：

```
//记录当前圆形所在九宫格的位置下标
private int creentTileIndex;
```

绘图函数：

```
public void myDraw() {
```

```
        ...
        canvas.drawColor(Color.WHITE);
        paint.setStyle(Style.STROKE);
        //绘制九宫格(将屏幕九等份)
        //得到每个方格的宽高
        int tileW = screenW / 3;
        int tileH = screenH / 3;
        for (int i = 0; i < 3; i++) {
                for (int j = 0; j < 3; j++) {
                    canvas.drawRect(i * tileW, j * tileH, (i + 1)* tileW,
                                        (j + 1) * tileH, paint);
                }
        }
        paint.setStyle(Style.FILL);
        //根据得到的圆形下标位置进行绘制相应的方格中
        canvas.drawCircle(creentTileIndex % 3 * tileW + tileW / 2,
                    creentTileIndex / 3 * tileH + tileH / 2, 30, paint);
        //操作说明
        canvas.drawText("上键：保存游戏", 0, 20, paint);
        canvas.drawText("下键：读取游戏", 110, 20, paint);
        canvas.drawText("左右键：移动圆形", 215, 20, paint);
        ...
}
```

到此小游戏完成，为保存数据做好了准备。项目运行效果如图4-59所示。

接下来，对圆形添加移动功能以及对游戏进行添加“存储和读取”功能。

首先添加一个成员变量：

```
private SharedPreferences sp;
```

然后修改构造函数：

```
public MySurfaceView(Context context) {
        ...
        //通过Context获取SharedPreference实例
        sp = context.getSharedPreferences("SaveName",
                                                Context.MODE_PRIVATE);
        //每次程序运行时获取圆形的下标
        int tempIndex = sp.getInt("CirCleIndex", -1);
        //判定如果返回-1 说明没有找到，就不对当前记录圆形的变量进行赋值
        if (tempIndex != -1) {
                creentTileIndex = tempIndex;
        }
        ...
}
```

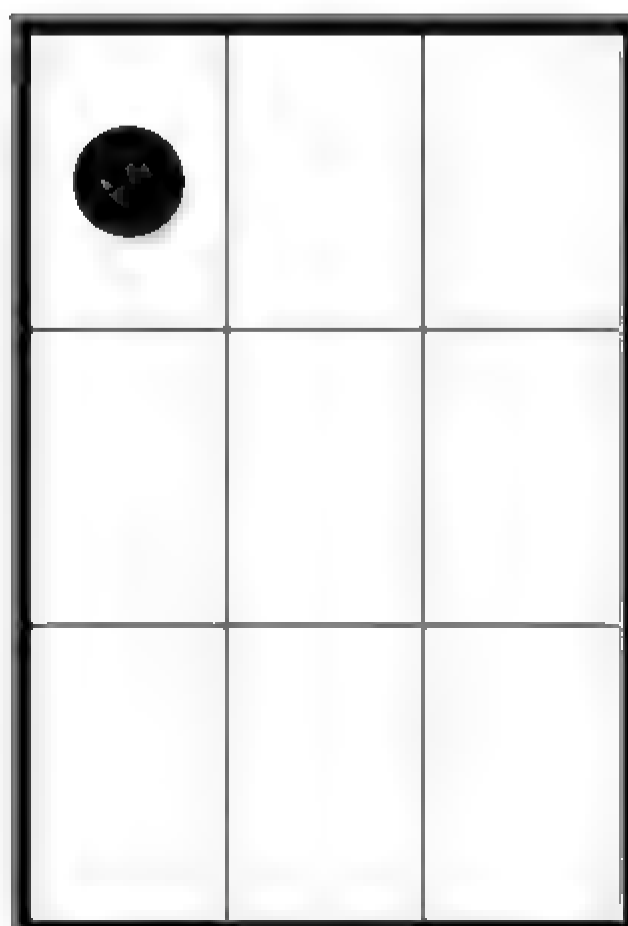


图 4-59 项目截图

在程序刚启动时就应该读取上次运行所在的下标值，如果取不到值，说明之前没有保存过，那么将不赋值给当前程序的圆形下标变量。

最后就是添加实体按键处理：

```
public boolean onKeyDown(int keyCode, KeyEvent event) {
        //上键保存游戏状态
        if (keyCode == KeyEvent.KEYCODE_DPAD_UP) {
                sp.edit().putInt("CirCleIndex", creentTileIndex).commit();
        //下键读取游戏状态
        } else if (keyCode == KeyEvent.KEYCODE_DPAD_DOWN) {
                int tempIndex = sp.getInt("CirCleIndex", -1);
                if (tempIndex != -1) {
                        creentTileIndex = tempIndex;
                }
        //圆形的移动
        } else if (keyCode == KeyEvent.KEYCODE_DPAD_LEFT) {
                if (creentTileIndex > 0) {
                        creentTileIndex -= 1;
                }
        } else if (keyCode == KeyEvent.KEYCODE_DPAD_RIGHT) {
                if (creentTileIndex < 8) {
                        creentTileIndex += 1;
                }
        }
        return super.onKeyDown(keyCode, event);
}
```

下面运行项目，并且将圆形移动到中间格中，单击上键保存其位置，然后重新运行项目观察效果，如图 4-60 所示。

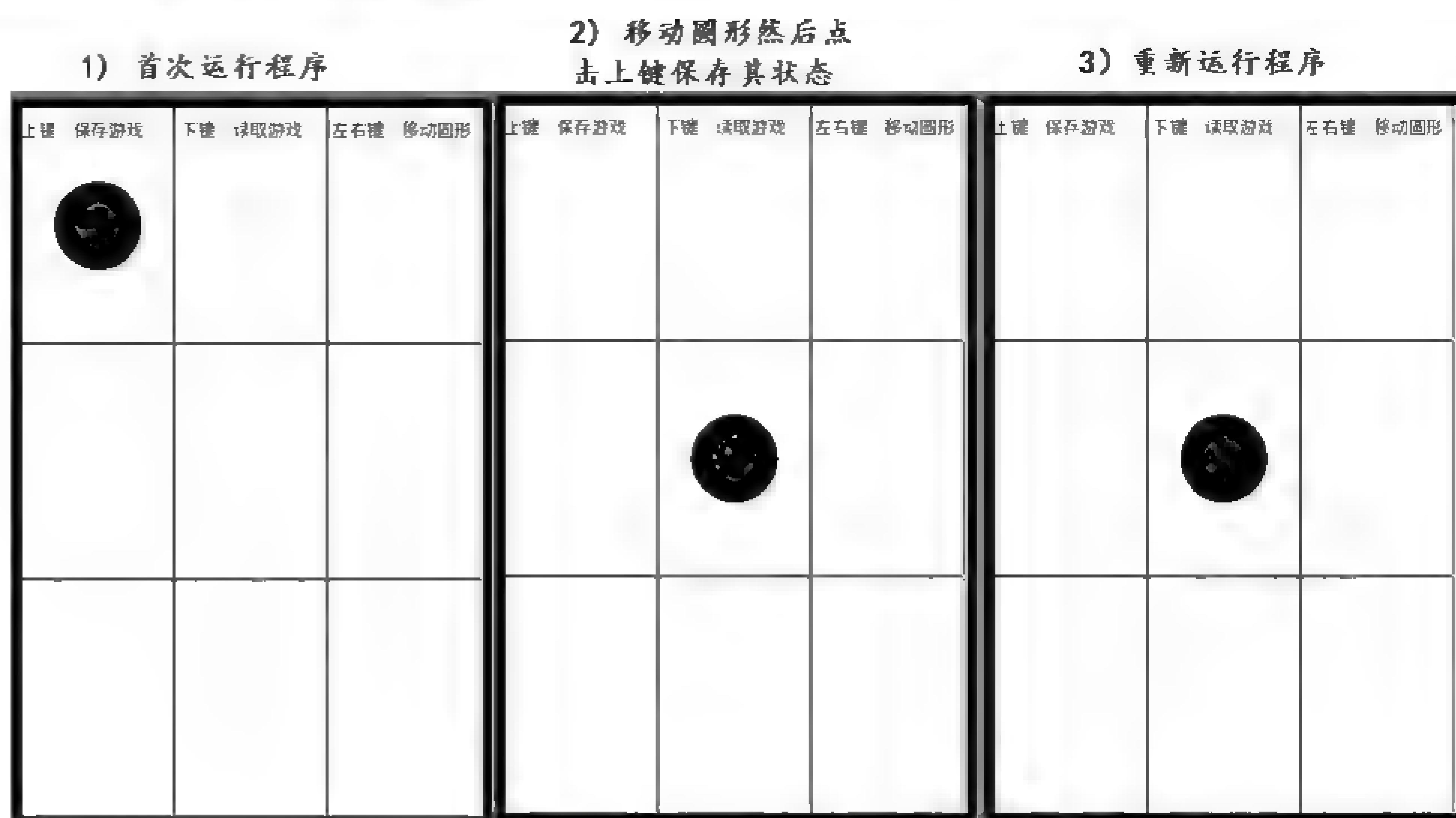


图 4-60 SharedPreference 存储

4.16.2 流文件存储

为了便于讲解，仍然利用上一小节的小游戏存储代码，但是去除 SharedPreference 存储部分，这里改用流文件形式进行保存，项目对应的源代码为“4-16-2（游戏保存之 Stream）”。本项目中，需要修改的只有游戏“保存”与“读取”的操作。

实体按键监听函数：

```
public boolean onKeyDown(int keyCode, KeyEvent event) {
        //用到的读出、写入流
        FileOutputStream fos = null;
        FileInputStream fis = null;
        DataOutputStream dos = null;
        DataInputStream dis = null;
        //上键保存游戏状态
        if (keyCode == KeyEvent.KEYCODE_DPAD_UP) {
                try {
                        //利用 Activity 实例打开流文件得到一个写入流
                        fos = MainActivity.instance.openFileOutput
                                    ("save.himi", Context.MODE_PRIVATE);
                        //将写入流封装在数据写入流中
                        dos = new DataOutputStream(fos);
                        //写入一个 int 类型(将圆形所在格子的下标写入流文件中)
                        dos.writeInt(creentTileIndex);
                } catch (FileNotFoundException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
```

```
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } finally {
            //即使保存时发生异常，也要关闭流
            try {
                if (fos != null)
                    fos.close();
                if (dos != null)
                    dos.close();
            } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
    //下键读取游戏状态
    } else if (keyCode == KeyEvent.KEYCODE_DPAD_DOWN) {
        try {
            if (MainActivity.instance.openFileInput
                    ("save.himi") != null) {
                try {
                    //利用Activity实例打开流文件得到一个读入流
                    fis = MainActivity.instance.
                            openFileInput("save.himi");
                    //将读入流封装在数据读入流中
                    dis = new DataInputStream(fis);
                    //读出一个Int类型赋值与圆形所在格子的下标
                    creentTileIndex = dis.readInt();
                } catch (FileNotFoundException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                } catch (IOException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                } finally {
                    //即使读取时发生异常，也要关闭流
                    try {
                        if (fis != null)
                            fis.close();
                        if (dis != null)
                            dis.close();
                    } catch (IOException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                    }
                }
```

```
                }
            } catch (FileNotFoundException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        //圆形的移动
        } else if (keyCode == KeyEvent.KEYCODE_DPAD_LEFT) {
            if (creentTileIndex > 0) {
                creentTileIndex -= 1;
            }
        } else if (keyCode == KeyEvent.KEYCODE_DPAD_RIGHT) {
            if (creentTileIndex < 8) {
                creentTileIndex += 1;
            }
        }
        return super.onKeyDown(keyCode, event);
}
```

不管是读入还是写入，都通过 Activity 打开流文件得到输入输出流。当需要写入流文件时，如果打开的流文件不存在，那么 Android 会自动生成对应的流文件；而当需要读入流文件时，首先应该判定流文件是否存在，一旦流文件不存在，就会抛出异常。

这里流形式的保存操作比较简单，需要注意的是：

- 读流时，一定要记得先判断是否存在需要操作的流文件；
- 写入和读入的数据类型要配对，顺序也不能错；例如：写入时，先写入了一个 boolean 值，然后又写入了一个 Int 值；那么读入时，也应该先读 boolean 类型，然后再读 Int 类型；
- 流一旦打开一定要关闭，为了避免流操作出现异常，需确保正常关闭流，应该将关闭操作写在 finally 语句中；
- file 流使用 Data 流进行了封装，这样做的原因是可以获得更多的操作方式，便于对数据的处理。

以上是使用流文件保存的方式，但是也只是将保存后的流文件默认放在了系统内存里。一般游戏的数据可能会有很多，所以不应该放在手机内存中，而是放在 SDCard 中，这样就不用担心系统因游戏保存的数据过多导致内存不足等问题。

将流文件保存在SDCard中的详细步骤如下：

（1）声明读入权限：

Android 中的一些操作，比如：读取通讯录信息、发送短信、使用联网、GPRS 等功能都需要在项目 AndroidManifest.xml 中声明使用权限，然后才可正常使用其功能。

当然在很多时候，是不知道是否需要声明添加权限的，其实这个也不用知道，因为如果用到这些需要声明权限的功能，且恰好没有声明的情况下，在LogCat中是会报异常的，其异常则提醒需要添加对应的权限。

写入权限如下：

```
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
```

（2）创建目录和存储文件

使用 SDCard 的方式进行存储数据，写入的时候 Android 不会跟存储系统默认路径那样默认生成存储文件，所以必须自己来创建；如果存储的文件有自定义路径的话，那么这个路径也需要手动添加。

假定存储文件在 SDCard 的路径为/sdcard/himi/save.himi。

①首先需要创建路径 /sdcard/himi

```
//声明一个路径
File path = new File("/sdcard/himi");
if (!path.exists()) {
      path.mkdirs();
}
```

M boolean File.exists()

作用：判断是否存在当前目录

返回值：当目录存在返回 true

M boolean File.mkdirs()

作用：创建一个目录

返回值：当创建成功返回 true

②然后创建存储文件：/sdcard/himi/save.himi

```
//声明文件路径
File f = new File("/sdcard/himi/save.himi");
if (!f.exists()) {// 文件存在返回 true
      f.createNewFile();// 创建一个文件
}
```

M boolean File.createNewFile()

作用：创建一个文件

返回值：创建成功返回 true

（3）通过加载指定路径的存储文件获取输入输出流

- 输入流：FileInputStream fis = new FileInputStream（File file）；
- 输出流：FileOutputStream fos = new FileOutputStream（File file）。

除此之外还需要知道一点，因为有时手机设备并没有安装 SDCard，或者当前 SDCard 处于被移除的状态时，为了避免这两种情况带来的异常，需要通过下面方法获取当前手机设备 SDCard 的状态：

String Environment.getExternalStorageState()

作用：获取当前 SDCard 的状态

返回值：当 SDCard 不存在时，返回 null；

当 SDCard 处于移除状态时，返回“removed”；

知道这些之后，本对之前使用流保存数据默认保存在手机内存里的方式进行添加修改，使之在默认手机设备存在 SDCard 时，保存在 SDCard 中；当 SDCard 不存在或者 SDCard 正处于被移除的状态时，默认将数据保存在手机内存中。

按键监听函数修改如下：

```
public boolean onKeyDown(int keyCode, KeyEvent event) {
    ...
    //用到的读出、写入流
    FileOutputStream fos = null;
    FileInputStream fis = null;
    DataOutputStream dos = null;
    DataInputStream dis = null;
    //上键保存游戏状态
    if (keyCode == KeyEvent.KEYCODE_DPAD_UP) {
        try {
            // 从SDcard中写入数据
            // 试探终端是否有sdcard! 并且探测SDCard是否处于被移除的状态
            if (Environment.getExternalStorageState() != null
&& !Environment.getExternalStorageState().equals("removed")) {
                Log.v("Himi", "写入, 有SD卡");
                File path = new File("/sdcard/himi");// 创建目录
                File f=new File("/sdcard/himi/save.himi");//创建文件
                if (!path.exists()) {// 目录存在返回true
                    path.mkdirs();// 创建一个目录
                }
                if (!f.exists()) {// 文件存在返回true
                    f.createNewFile();// 创建一个文件
                }
                fos = new FileOutputStream(f);// 将数据存入sd卡中
            } else {
                //默认系统路径
                //利用Activity实例打开流文件得到一个写入流
                fos = MainActivity.instance.openFileOutput ("save.himi",
                    Context.MODE_PRIVATE);
                }
                //将写入流封装在数据写入流中
                dos = new DataOutputStream(fos);
                //写入一个int类型(将圆形所在格子的下标写入流文件中)
                dos.writeInt(creentTileIndex);
        } catch (FileNotFoundException e) {
            // TODO Auto-generated catch block
```

```
            e.printStackTrace();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } finally {
            //即使保存时发生异常，也要关闭流
            try {
                if (fos != null)
                    fos.close();
                if (dos != null)
                    dos.close();
            } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
    //下键读取游戏状态
    } else if (keyCode == KeyEvent.KEYCODE_DPAD_DOWN) {
            boolean isHaveSDCard = false;
            // 从 SDcard 中读取数据
            // 试探终端是否有 sdcard! 并且探测 SDCard 是否处于被移除的状态
            if (Environment.getExternalStorageState() != null
&& !Environment.getExternalStorageState().equals("removed")) {
                Log.v("Himi", "读取，有 SD 卡");
                isHaveSDCard = true;
            }
            try {
                if (isHaveSDCard) {
                    File path = new File("/sdcard/himi");// 创建目录
                    File f=new File("/sdcard/himi/save.himi");//创建文件
                    if (!path.exists()) {// 目录存在返回 true
                        return false;
                    } else {
                        if (!f.exists()) {// 文件存在返回 true
                            return false;
                        }
                    }
                    fis = new FileInputStream(f);// 将数据存入 sd 卡中
                } else {
                    if
(MainActivity.instance.openFileInput("save.himi") !=
null) {
                        //利用 Activity 实例打开流文件得到一个读入流
                        fis = MainActivity.instance.openFileInput
                        ("save.himi");
                    }
```

```
                }
                //将读入流封装在数据读入流中
                dis = new DataInputStream(fis);
                //读出一个 Int 类型赋值与圆形所在格子的下标
                creentTileIndex = dis.readInt();
            } catch (FileNotFoundException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            } finally {
                //即使读取时发生异常，也要关闭流
                try {
                    if (fis != null)
                        fis.close();
                    if (dis != null)
                        dis.close();
                } catch (IOException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
        }
        ...
}
```

4.16.3 SQLite

SQLite 是一款轻量级数据库，它的设计目的是用于嵌入式系统，而且它占用的系统资源非常少，只有几百 KB。

SQLite具有如下特性：

- 轻量级。使用 SQLite 只需要带一个动态库，就可以享受它的全部功能，而且动态库的尺寸相当小。
- 独立性。SQLite 数据库的核心引擎不需要依赖第三方软件，也不需要所谓的“安装”。
- 隔离性。SQLite 数据库中所有的信息（比如表、视图、触发器等）都包含在一个文件夹内，方便管理和维护。
- 跨平台。SQLite 目前支持大部分操作系统，不只是电脑操作系统，在众多的手机系统中也是能够运行的，比如：Android。
- 多语言接口。SQLite 数据库支持多语言编程接口。
- 安全性。SQLite 数据库通过数据库级上的独占性和共享锁来实现独立事务处理。这意

味着多个进程可以在同一时间从同一数据库读取数据，但只有一个可以写入数据。

此种存储方式比较灵活，而且更加适合大数据量游戏的存储，当然利用轻量级数据库SQLite也可以存储到SDCard中。

由于SQLite存储方式涉及到SQL语言的基础知识，比如一些基础的对数据库的操作、删除、添加、修改等语句。对于没有数据库基础的读者，这里讲解的太过简单，如果对SQLite感兴趣的话，可以参考其他资料与书籍，也可以登录作者的博客进行学习。

SQLite存储的内容可以参考以下地址：

http://blog.csdn.net/xiaominghimi/archive/2011/01/04/6114629.aspx

4.17 本章小节

本章介绍了 Android 游戏开发的基础知识，具体内容包括 Android 游戏开发中常用的三种视图，View、surface View 游戏框架及区别，Canvas 画布，Paint 画笔，Bitmap 位图的渲染与操作，剪切区域，动画，游戏适屏的简述与作用，让游戏主角动起来，碰撞检测，游戏音乐与音效，游戏数据存储等。这些内容是 Android 游戏开发人员必须掌握的。

第 5 章

游戏开发实战演练

从本章节可以学习到：

- ❖ 项目前的准备工作
- ❖ 划分游戏状态
- ❖ 游戏初始化（菜单界面）
- ❖ 游戏界面
- ❖ 游戏胜利与结束界面
- ❖ 游戏细节处理



通过上一章对 Android 平台游戏开发基础的学习，大家应该对游戏视图的框架、基本的绘图、图形的渲染、游戏音效、背景音乐以及游戏存储都有一定的掌握；虽然这些很基础，但足够用来独立开发游戏应用。

本章主要通过一个“飞行射击”类型的实战项目，让大家一方面掌握游戏开发的流程，深入学习在一款游戏中模块之间是如何相辅相成并完成一款游戏的。另一方面通过实战项目发现一些不经意的细节问题及编程缺陷。

5.1 项目前的准备工作

新建项目“PlaneGame”，游戏框架为 SurfaceView 游戏框架，项目对应的源代码为“5-1（飞行射击游戏实战）”。项目使用的图片资源如图 5-1 所示。

资源文件与名称	尺寸(像素)	备注	资源文件与名称	尺寸(像素)	备注
gamelost.png	320*480	游戏失败界面	gamewin.png	320*480	游戏胜利界面
background.png	320*683	游戏背景图	menu.png	320*480	游戏菜单界面
button.png	169*53	开始按钮(未按下)	button_press.png	160*51	开始按钮(按下)
	47*56	主角	enemy_pig.png	540*41	Boss(十帧)
enemy_duck.png	450*31	怪物鸭(十帧)	enemy_fly.png	1200*72	怪物苍蝇(十帧)
boom.png	308*49	爆炸效果七帧	boos_boom.png	210*43	BOSS爆炸效果(五帧)
hp.png	35*33	主角血量	bullet.png	17*29	主角飞机子弹
boosbullet.png	43*43	Boss 子弹	bullet_enemy.png	20*20	敌机子弹

图 5-1 “PlaneGame”项目使用的图片资源

5.2 划分游戏状态

为了让整个游戏的思路与代码更清晰，游戏一般都会定义一些常量表示当前游戏的状态；比如：游戏菜单界面、游戏界面、游戏胜利界面、游戏失败界面等等；整个游戏划分的状态，对应到代码中其实就是逻辑函数、按键监听事件、触屏监听事件以及绘制函数部分；当然游戏的图片加载也可以根据不同的状态进行动态加载，但是由于当前项目的图量不是很多，所以就不再动态加载。

定义游戏状态常量：

```
//定义游戏状态常量
public static final int GAME_MENU = 0;//游戏菜单
public static final int GAMEING = 1;//游戏中
public static final int GAME_WIN = 2;//游戏胜利
public static final int GAME_LOST = 3;//游戏失败
public static final int GAME_PAUSE = -1;//游戏菜单
//当前游戏状态(默认初始在游戏菜单界面)
public static int gameState = GAME_MENU;
```

绘图函数：

```
public void myDraw() {
        ...
        //绘图函数根据游戏状态不同进行不同绘制
        switch (gameState) {
        case GAME_MENU:
                break;
        case GAMEING:
                break;
        case GAME_PAUSE:
                break;
        case GAME_WIN:
                break;
        case GAME_LOST:
                break;
        }
        ...
}
```

实体按键按下监听函数：

```
public boolean onKeyDown(int keyCode, KeyEvent event) {
```

```
        //按键监听事件函数根据游戏状态不同进行不同监听
        switch (gameState) {
        case GAME_MENU:
                break;
        case GAMEING:
                break;
        case GAME_PAUSE:
                break;
        case GAME_WIN:
                break;
        case GAME_LOST:
                break;
        }
}
```

实体按键抬起监听函数：

```
public boolean onKeyUp(int keyCode, KeyEvent event) {
        //按键监听事件函数根据游戏状态不同进行不同监听
        switch (gameState) {
        case GAME_MENU:
                break;
        case GAMEING:
                break;
        case GAME_PAUSE:
                break;
        case GAME_WIN:
                break;
        case GAME_LOST:
                break;
        }
}
```

触屏事件监听函数:

```
public boolean onTouchEvent(MotionEvent event) {
        //触屏监听事件函数根据游戏状态不同进行不同监听
        switch (gameState) {
        case GAME_MENU:
                break;
        case GAMEING:
                break;
        case GAME_PAUSE:
                break;
        case GAME_WIN:
                break;
```

```
        case GAME_LOST:
            break;
        }
        return true;
}
```

逻辑函数：

```
private void logic() {
        ...
        //逻辑处理根据游戏状态不同进行不同处理
        switch (gameState) {
        case GAME_MENU:
            break;
        case GAMEING:
            break;
        case GAME_PAUSE:
            break;
        case GAME_WIN:
            break;
        case GAME_LOST:
            break;
        }
        ...
}
```

由于当前游戏素材是针对游戏竖屏状态下进行的，所以设置当前 Activity 保持竖屏：在 AndroidManifest.xml 文件中：

```
android:screenOrientation="portrait"
```

5.3 游戏初始化（菜单界面）

游戏的初始化，比如加载图片等一般都放在视图创建函数中进行，是因为视图宽高只有在视图创建中才能正常获取到，一些图片坐标等都要根据视图宽高进行设置（考虑适屏）。

除此之外，考虑到游戏胜利或者失败后需要重新进入游戏等因素，应该添加一个自定义函数来完成游戏初始化工作；一旦需要重置游戏，只要调用此函数即可。

```
//声明一个Resources 实例便于加载图片
private Resources res = this.getResources();
//声明游戏需要用到的图片资源(图片声明)
private Bitmap bmpBackGround;//游戏背景
```

```
private Bitmap bmpBoom;//爆炸效果
private Bitmap bmpBoosBoom;//Boos 爆炸效果
private Bitmap bmpButton;//游戏开始按钮
private Bitmap bmpButtonPress;//游戏开始按钮被点击
private Bitmap bmpEnemyDuck;//怪物鸭子
private Bitmap bmpEnemyFly;//怪物苍蝇
private Bitmap bmpEnemyBoos;//怪物猪头 Boos
private Bitmap bmpGameWin;//游戏胜利背景
private Bitmap bmpGameLost;//游戏失败背景
private Bitmap bmpPlayer;//游戏主角飞机
private Bitmap bmpPlayerHp;//主角飞机血量
private Bitmap bmpMenu;//菜单背景
public static Bitmap bmpBullet;//子弹
public static Bitmap bmpEnemyBullet;//敌机子弹
public static Bitmap bmpBossBullet;//Boss 子弹
```

视图创建函数：

```
public void surfaceCreated(SurfaceHolder holder) {
      ...
      initGame();//便于初始化游戏
      ...
}
```

自定义 initGame 函数：

```
private void initGame() {
      //放置游戏切入后台重新进入游戏时，游戏被重置!
      //当游戏状态处于菜单时，才会重置游戏
      if (gameState == GAME_MENU) {
            //加载游戏资源
            bmpBackGround = BitmapFactory.decodeResource(res, R.drawable
                           .background);
            bmpBoom = BitmapFactory.decodeResource(res, R.drawable.boom);
            bmpBoosBoom = BitmapFactory.decodeResource(res, R.drawable
                           .boos_boom);
            bmpButton = BitmapFactory.decodeResource(res, R.drawable
                            .button);
            bmpButtonPress = BitmapFactory.decodeResource(res, R.drawable
                            .button_press);
            bmpEnemyDuck = BitmapFactory.decodeResource(res, R.drawable
                           .enemy_duck);
            bmpEnemyFly = BitmapFactory.decodeResource(res, R.drawable
                            .enemy_fly);
            bmpEnemyBoos = BitmapFactory.decodeResource(res, R.drawable
                            .enemy_pig);
            bmpGameWin - BitmapFactory.decodeResource(res, R.drawable
```

```
                    .gamewin);
            bmpGameLost = BitmapFactory.decodeResource(res, R.drawable
                    .gamelost);
            bmpPlayer = BitmapFactory.decodeResource(res, R.drawable
                    .player);
            bmpPlayerHp = BitmapFactory.decodeResource(res,
                    R.drawable.hp);
            bmpMenu = BitmapFactory.decodeResource(res, R.drawable.menu);
            bmpBullet = BitmapFactory.decodeResource(res, R.drawable
                    .bullet);
            bmpBullet = BitmapFactory.decodeResource(res, R.drawable
                    .bullet);
            bmpBossBullet = BitmapFactory.decodeResource(res, R.drawable
                    .boosbullet);
      }
}
```

当然这里只是简单的初始化了游戏资源图而已，对于游戏状态初始化应该是菜单界面，下面来新建一个菜单类：每个界面或者每个功能单独提出来做一个类，让其拥有自己的逻辑，绘图等函数，那么在主游戏类 MySurfaceView 中代码和思路都会清晰很多；但并不是所有的游戏都要做一个界面，这些要根据具体项目而定。

新建类 GameMenu（菜单界面）的代码如下：

```
public class GameMenu {
      //菜单背景图
      private Bitmap bmpMenu;
      //按钮图片资源(按下和未按下图)
      private Bitmap bmpButton, bmpButtonPress;
      //按钮的坐标
      private int btnX, btnY;
      //按钮是否按下标识位
      private Boolean isPress;
      //菜单初始化
      public GameMenu(Bitmap bmpMenu, Bitmap bmpButton, Bitmap
bmpButtonPress) {
            this.bmpMenu = bmpMenu;
            this.bmpButton = bmpButton;
            this.bmpButtonPress = bmpButtonPress;
            //X 居中，Y 紧接屏幕底部
            btnX = MySurfaceView.screenW / 2 - bmpButton.getWidth() / 2;
            btnY = MySurfaceView.screenH - bmpButton.getHeight();
            isPress = false;
      }
      //菜单绘图函数
      public void draw(Canvas canvas, Paint paint) {
```

```
            //绘制菜单背景图
            canvas.drawBitmap(bmpMenu, 0, 0, paint);
            //绘制未按下按钮图
            if (isPress) {//根据是否按下绘制不同状态的按钮图
                canvas.drawBitmap(bmpButtonPress, btnX, btnY, paint);
            } else {
                canvas.drawBitmap(bmpButton, btnX, btnY, paint);
            }
        }
        //菜单触屏事件函数，主要用于处理按钮事件
        public void onTouchEvent(MotionEvent event) {
            //获取用户当前触屏位置
            int pointX = (int) event.getX();
            int pointY = (int) event.getY();
            //当用户是按下动作或移动动作
            if (event.getAction() == MotionEvent.ACTION_DOWN ||
event.getAction() == MotionEvent.ACTION_MOVE) {
                //判定用户是否点击了按钮
                if (pointX > btnX && pointX < btnX +
bmpButton.getWidth()) {
                    if (pointY > btnY && pointY < btnY +
bmpButton.getHeight()) {
                        isPress = true;
                    } else {
                        isPress = false;
                    }
                } else {
                    isPress = false;
                }
                //当用户是抬起动作
            } else if (event.getAction() == MotionEvent.ACTION_UP) {
                //抬起判断是否点击按钮，防止用户移动到别处
                if (pointX > btnX && pointX < btnX +
bmpButton.getWidth()) {
                    if (pointY > btnY && pointY < btnY +
bmpButton.getHeight()) {
                        //还原 Button 状态为未按下状态
                        isPress = false;
                        //改变当前游戏状态为开始游戏
                        MySurfaceView.gameState =
MySurfaceView.GAMEING;
                    }
                }
            }
        }
}
```

从菜单类中可以看到它拥有自己的绘图、触屏事件处理函数。每个类的设计封装，不是一开始就能想的很全面，它是在不断修改项目，不断发现和添加中完善的。所以类的封装，一开始只需要将能想到的基本属性写上即可，比如此菜单类，肯定能想到的是它有背景，并且有按钮、那么按钮则需要图片资源、坐标等属性。其他的可能想不到，不过随着项目不断完善，需要用到时就会自然而然的想到，届时再去添加完善类即可，千万不要一开始就浪费大量时间去设计类。

菜单类完成后，主视图类 MySurfaceView 的绘图和触屏事件交给菜单自己去处理就可以了。

MySurfaceView 类修改如下：

```
//声明一个菜单对象
private GameMenu gameMenu;
```

初始化函数：

```
private void initGame() {
	...
	//菜单类实例
	gameMenu = new GameMenu(bmpMenu, bmpButton, bmpButtonPress);
	...
}
```

绘图函数：

```
public void myDraw() {
	...
	switch (gameState) {
	case GAME_MENU:
		//菜单的绘图函数
		gameMenu.draw(canvas, paint);
		break;
	case GAMEING:
		break;
	case GAME_PAUSE:
		break;
	case GAME_WIN:
		break;
	case GAME_LOST:
		break;
	}
	...
}
```

触屏监听事件：

```
public boolean onTouchEvent(MotionEvent event) {
```

```
        //触屏监听事件函数根据游戏状态不同进行不同监听
        switch (gameState) {
        case GAME_MENU:
                //菜单的触屏事件处理
                gameMenu.onTouchEvent(event);
                break;
        case GAMEING:
                break;
        case GAME_PAUSE:
                break;
        case GAME_WIN:
                break;
        case GAME_LOST:
                break;
        }
        return true;
}
```

此时运行项目，效果如图5-2所示。

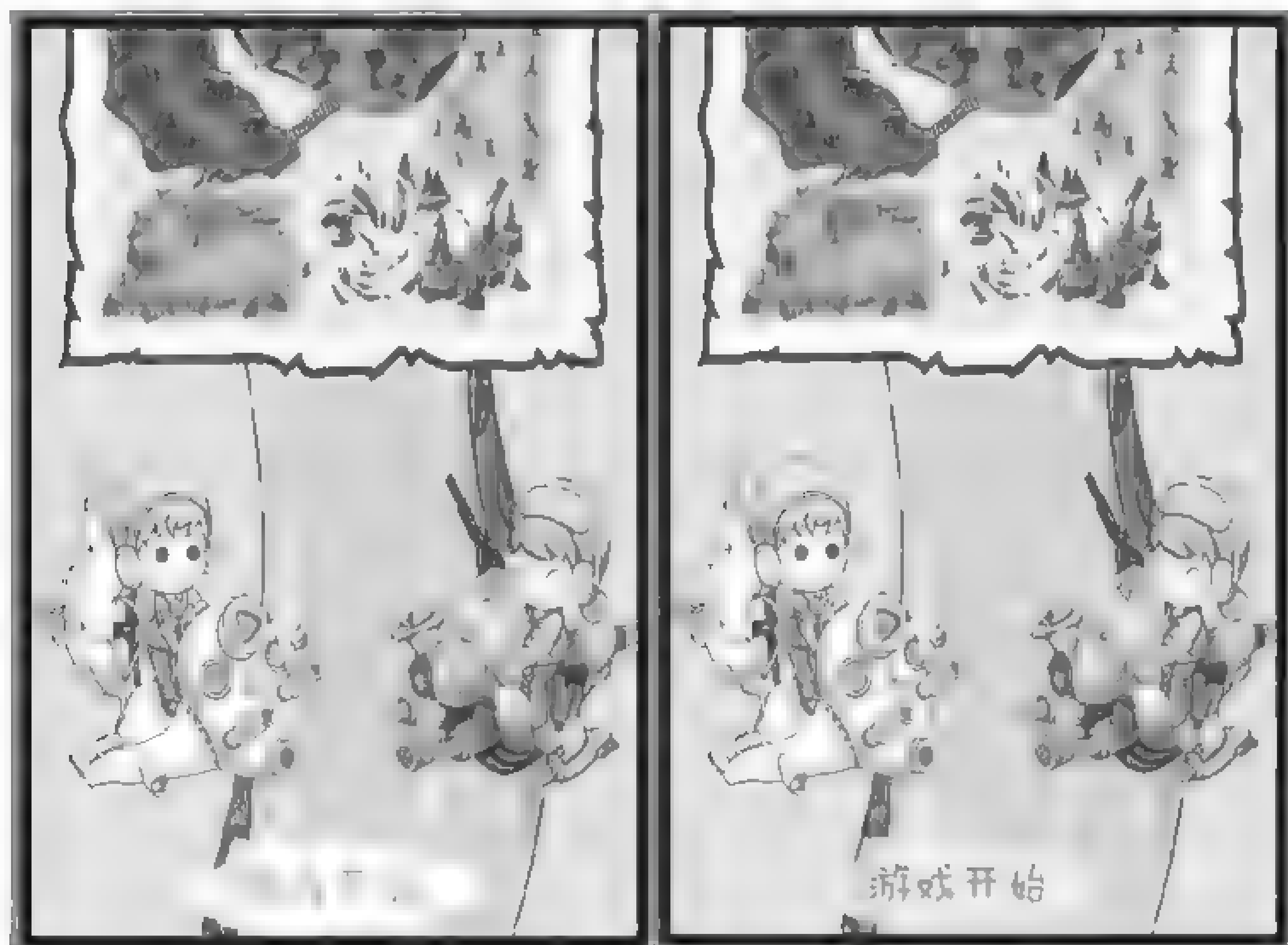


图5-2　菜单界面（右侧图按钮为点击后的效果）

5.4 游戏界面

当用户单击菜单中的“游戏开始”按钮后，即可进入游戏界面。首先分析整个游戏中包含的元素：滚动的游戏背景、可操作的主角、主角的子弹、主角的血量、两种怪物（敌机）

和一个 Boss、以及敌机和 Boss 的爆炸效果。

5.4.1 实现滚动的背景图

如果想让游戏背景更加绚丽，可以让背景有多层，让每一层移动速度均不同；还有一些游戏背景，比如 RPG、ARPG 等类型的游戏，都是由小图片拼成，实现方式是一张图，有多帧图块，通过事先定义好的帧下标数组，按照其数组的下标绘制对应图块帧即可；本项目中简单的实现循环播放背景图。

新建类 GameBg 的代码如下：

```
public class GameBg {
	//游戏背景的图片资源
	//为了循环播放，这里定义两个位图对象，
	//其资源引用的是同一张图片
	private Bitmap bmpBackGround1;
	private Bitmap bmpBackGround2;
	//游戏背景坐标
	private int bg1x, bg1y, bg2x, bg2y;
	//背景滚动速度
	private int speed = 3;

	//游戏背景构造函数
	public GameBg(Bitmap bmpBackGround) {
		this.bmpBackGround1 = bmpBackGround;
		this.bmpBackGround2 = bmpBackGround;
		//首先让第一张背景底部正好填满整个屏幕
		bg1y = -Math.abs(bmpBackGround1.getHeight() -
				MySurfaceView.screenH);
		//第二张背景图紧接在第一张背景的上方
		//+101 的原因：虽然两张背景图无缝隙连接但是因为图片资源头尾
		//直接连接不和谐，为了让视觉看不出是两张图连接而修正的位置
		bg2y = bg1y - bmpBackGround1.getHeight() + 111;
	}
	//游戏背景的绘图函数
	public void draw(Canvas canvas, Paint paint) {
		//绘制两张背景
		canvas.drawBitmap(bmpBackGround1, bg1x, bg1y, paint);
		canvas.drawBitmap(bmpBackGround2, bg2x, bg2y, paint);
	}
	//游戏背景的逻辑函数
	public void logic() {
		bg1y += speed;
		bg2y += speed;
		//当第一张图片的 Y 坐标超出屏幕，
```

```
            //立即将其坐标设置到第二张图的上方
            if (bg1y > MySurfaceView.screenH) {
                bg1y = bg2y - bmpBackGround1.getHeight() + 111;
            }
            //当第二张图片的Y坐标超出屏幕，
            //立即将其坐标设置到第一张图的上方
            if (bg2y > MySurfaceView.screenH) {
                bg2y = bg1y - bmpBackGround1.getHeight() + 111;
            }
        }
}
```

5.4.2 实现主角以及与主角相关的元素

本小节实现主角以及与主角相关的元素。新建类Player的代码如下：

```
public class Player {
    //主角的血量与血量位图
    //默认3血
    private int playerHp = 3;
    private Bitmap bmpPlayerHp;
    //主角的坐标以及位图
    public int x, y;
    private Bitmap bmpPlayer;
    //主角移动速度
    private int speed = 5;
    //主角移动标识（基础章节已讲解，你懂得）
    private boolean isUp, isDown, isLeft, isRight;
    //主角的构造函数
    public Player(Bitmap bmpPlayer, Bitmap bmpPlayerHp) {
        this.bmpPlayer = bmpPlayer;
        this.bmpPlayerHp = bmpPlayerHp;
        x = MySurfaceView.screenW / 2 - bmpPlayer.getWidth() / 2;
        y = MySurfaceView.screenH - bmpPlayer.getHeight();
    }
    //主角的绘图函数
    public void draw(Canvas canvas, Paint paint) {
        //绘制主角
        canvas.drawBitmap(bmpPlayer, x, y, paint);
        //绘制主角血量
        for (int i = 0; i < playerHp; i++) {
            canvas.drawBitmap(bmpPlayerHp, i * bmpPlayerHp.getWidth(),
            MySurfaceView.screenH - bmpPlayerHp.getHeight(), paint);
        }
    }
```

```
//实体按键
public void onKeyDown(int keyCode, KeyEvent event) {
        if (keyCode == KeyEvent.KEYCODE_DPAD_UP) {
                isUp = true;
        }
        if (keyCode == KeyEvent.KEYCODE_DPAD_DOWN) {
                isDown = true;
        }
        if (keyCode == KeyEvent.KEYCODE_DPAD_LEFT) {
                isLeft = true;
        }
        if (keyCode == KeyEvent.KEYCODE_DPAD_RIGHT) {
                isRight = true;
        }
}
//实体按键抬起
public void onKeyUp(int keyCode, KeyEvent event) {
        if (keyCode == KeyEvent.KEYCODE_DPAD_UP) {
                isUp = false;
        }
        if (keyCode == KeyEvent.KEYCODE_DPAD_DOWN) {
                isDown = false;
        }
        if (keyCode == KeyEvent.KEYCODE_DPAD_LEFT) {
                isLeft = false;
        }
        if (keyCode == KeyEvent.KEYCODE_DPAD_RIGHT) {
                isRight = false;
        }
}
//主角的逻辑
public void logic() {
        //处理主角移动
        if (isLeft) {
                x -= speed;
        }
        if (isRight) {
                x += speed;
        }
        if (isUp) {
                y -= speed;
        }
        if (isDown) {
                y += speed;
        }
        //判断屏幕X边界
```

```
            if (x + bmpPlayer.getWidth() >  MySurfaceView.screenW) {
                  x = MySurfaceView.screenW - bmpPlayer.getWidth();
            } else if (x <= 0) {
                  x = 0;
            }
            //判断屏幕Y边界
            if (y + bmpPlayer.getHeight() >= MySurfaceView.screenH) {
                  y = MySurfaceView.screenH - bmpPlayer.getHeight();
            } else if (y <= 0) {
                  y = 0;
            }
      }
      //设置主角血量
      public void setPlayerHp(int hp) {
            this.playerHp = hp;
      }
      //获取主角血量
      public int getPlayerHp() {
            return playerHp ;
      }
}
```

背景和主角类完成已初步完成，需要添加到主视图 MySurfaceView 类中。

首先声明两个类的成员对象：

```
//声明一个滚动游戏背景对象
private GameBg backGround;
//声明主角对象
private Player player;
```

初始化游戏函数：

```
private void initGame() {
      ...
      //实例游戏背景
      backGround = new GameBg(bmpBackGround);
      //实例主角
      player = new Player(bmpPlayer, bmpPlayerHp);
      ...
}
```

绘图函数：

```
public void myDraw() {
```

```
    ...
    switch (gameState) {
    case GAME_MENU:
        ...
        break;
    case GAMEING:
        //游戏背景
        backGround.draw(canvas, paint);
        //主角绘图函数
        player.draw(canvas, paint);
        break;
    case GAME_PAUSE:
        break;
    case GAME_WIN:
        break;
    case GAME_LOST:
        break;
    }
    ...
}
```

实体按键按下监听函数:

```
public boolean onKeyDown(int keyCode, KeyEvent event) {
    //按键监听事件函数根据游戏状态不同进行不同监听
    switch (gameState) {
    case GAME_MENU:
        break;
    case GAMEING:
        //主角的按键按下事件
        player.onKeyDown(keyCode, event);
        break;
    case GAME_PAUSE:
        break;
    case GAME_WIN:
        break;
    case GAME_LOST:
        break;
    }
    return super.onKeyDown(keyCode, event);
}
```

实体按键抬起监听函数:

```
public boolean onKeyUp(int keyCode, KeyEvent event) {
    //按键监听事件函数根据游戏状态不同进行不同监听
    switch (gameState) {
```

```
        case GAME_MENU:
            break;
        case GAMEING:
            //按键抬起事件
            player.onKeyUp(keyCode, event);
            break;
        case GAME_PAUSE:
            break;
        case GAME_WIN:
            break;
        case GAME_LOST:
            break;
        }
        return super.onKeyDown(keyCode, event);
}
```

逻辑函数：

```
private void logic() {
        //逻辑处理根据游戏状态不同进行不同处理
        switch (gameState) {
        case GAME_MENU:
            break;
        case GAMEING:
            //背景逻辑
            backGround.logic();
            //主角逻辑
            player.logic();
            break;
        case GAME_PAUSE:
            break;
        case GAME_WIN:
            break;
        case GAME_LOST:
            break;
        }
}
```

运行当前项目，效果如图 5-3 所示。

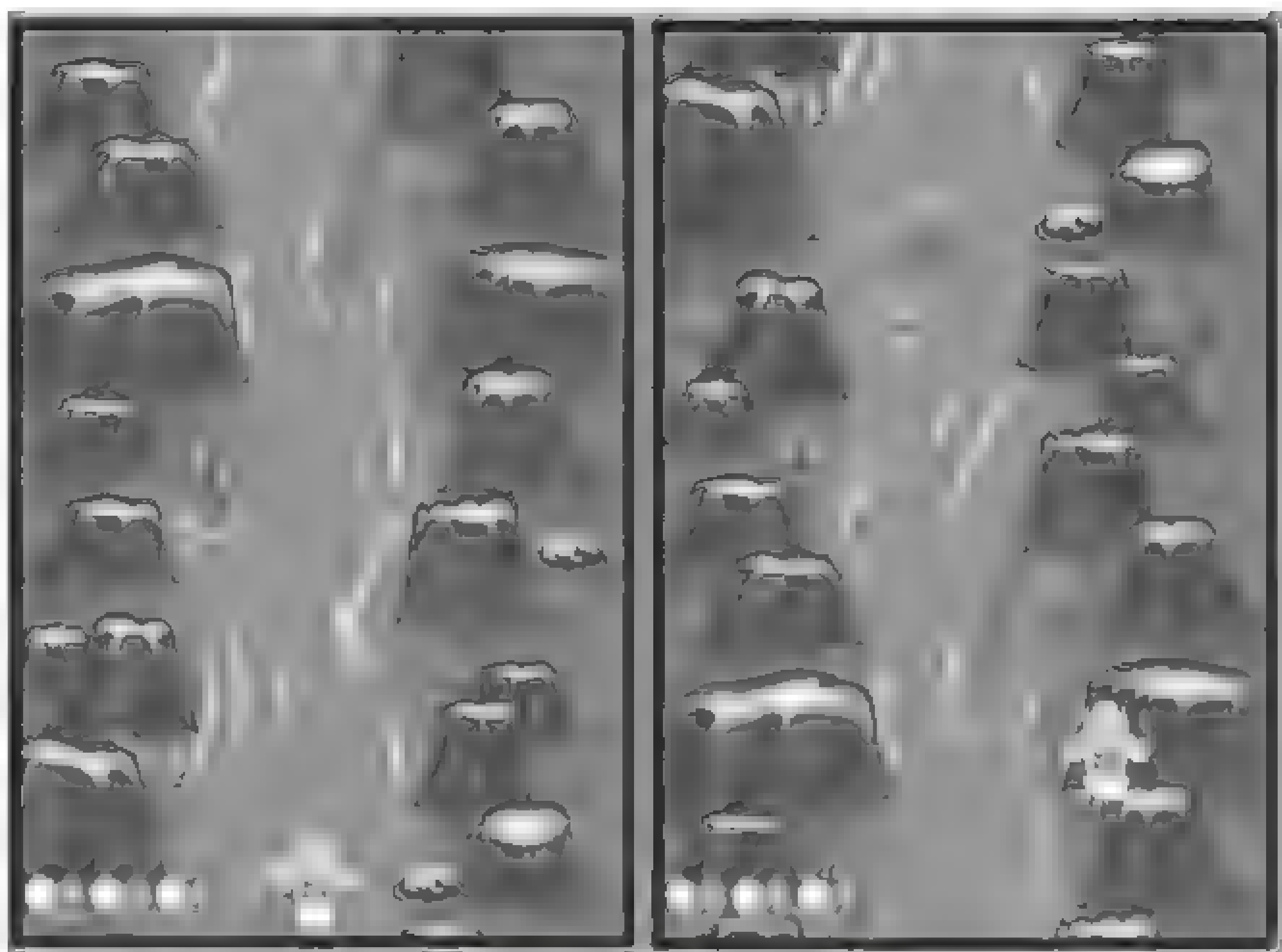


图 5-3　游戏背景与主角

5.4.3　怪物（敌机）类的实现

怪物类不同于主角，它的移动是不受玩家控制的，拥有自己的 AI（逻辑），而且它的朝向是朝下，主角的运动则是朝上，并且主角只有一个，而怪物则有多个；所以怪物类的封装至少会拥有一个类型属性，用以判定当前怪物的种类，毕竟每个怪物可能血量不同，AI 不同等等，所以添加一个类型属性来便于区分。

新建类 Enemy 的代码如下：

```
public class Enemy {
      //敌机的种类标识
      public int type;
      //苍蝇
      public static final int TYPE_FLY = 1;
      //鸭子(从左往右运动)
      public static final int TYPE_DUCKL = 2;
      //鸭子(从右往左运动)
      public static final int TYPE_DUCKR = 3;
      //敌机图片资源
      public Bitmap bmpEnemy;
      //敌机坐标
      public int x, y;
      //敌机每帧的宽高
      private int frameW, frameH;
      //敌机当前帧下标
      private int frameIndex;
      //敌机的移动速度
      private int speed;;
```

```
//判断敌机是否已经出屏
public boolean isDead;
//敌机的构造函数
public Enemy(Bitmap bmpEnemy, int enemyType, int x, int y) {
        this.bmpEnemy = bmpEnemy;
        frameW = bmpEnemy.getWidth() / 10;
        frameH = bmpEnemy.getHeight();
        this.type = enemyType;
        this.x = x;
        this.y = y;
        //不同种类的敌机速度不同
        switch (type) {
        //苍蝇
        case TYPE_FLY:
                speed = 25;
                break;
        //鸭子
        case TYPE_DUCKL:
                speed = 3;
                break;
        case TYPE_DUCKR:
                speed = 3;
                break;
        }
}
//敌机绘图函数
public void draw(Canvas canvas, Paint paint) {
        canvas.save();
        canvas.clipRect(x, y, x + frameW, y + frameH);
        canvas.drawBitmap(bmpEnemy, x - frameIndex * frameW,y, paint);
        canvas.restore();
}
//敌机逻辑 AI
public void logic() {
        //不断循环播放帧形成动画
        frameIndex++;
        if (frameIndex >= 10) {
                frameIndex = 0;
        }
        //不同种类的敌机拥有不同的 AI 逻辑
        switch (type) {
        case TYPE_FLY:
                if (isDead == false) {
                        //减速出现，加速返回
                        speed -= 1;
                        y += speed;
```

```
                if (y <= -200) {
                    isDead = true;
                }
            }
            break;
        case TYPE_DUCKL:
            if (isDead == false) {
                //斜右下角运动
                x += speed / 2;
                y += speed;
                if (x > MySurfaceView.screenW) {
                    isDead = true;
                }
            }
            break;
        case TYPE_DUCKR:
            if (isDead == false) {
                //斜左下角运动
                x -= speed / 2;
                y += speed;
                if (x < -50) {
                    isDead = true;
                }
            }
            break;
        }
    }
}
```

不论是敌机还是主角都拥有一个设置血量的函数，这也是为了后续添加敌机与主角、敌机子弹与主角之间碰撞处理留下的接口；当然现在也可以不写，毕竟当前还没有用到此函数的地方。

敌机类设计完毕之后，将其逻辑和绘制都添加到主视图 MySurfaceView 中：

```
    //声明一个敌机容器
    private Vector<Enemy> vcEnemy;
    //每次生成敌机的时间(毫秒)
    private int createEnemyTime = 50;
    private int count;//计数器
    //敌人数组：1 和 2 表示敌机的种类，-1 表示 Boss
    //二维数组的每一维都是一组怪物
    private int enemyArray[][] = {
                { 1, 2 }, { 1, 1 }, { 1, 3, 1, 2 }, { 1, 2 },
                { 2, 3 }, { 3, 1, 3 }, { 2, 2 }, { 1, 2 }, { 2, 2 },
                { 1, 3, 1, 1 },{ 2, 1 },{ 1, 3 }, { 2, 1 }, { -1 } };
    //当前取出一维数组的下标
```

```java
    private int enemyArrayIndex;
    //是否出现Boss标识位
    private boolean isBoss;
    //随机库，为创建的敌机赋予随即坐标
    private Random random;
```

这里定义的二维数组，其中每一维是每次创建敌机时的敌机种类和数量；这样定义主要是便于管理游戏敌机数量和难度以及游戏时间等。

一般在游戏开发中会将其数组写成流文件进行保存，以流文件的方式保存一方面仍是为了便于管理，另一方面对于关卡设计简单的做了一个加密。

游戏初始化函数：

```java
private void initGame() {
    ...
    //实例敌机容器
    vcEnemy = new Vector<Enemy>();
    //实例随机库
    random = new Random();
    ...
}
```

绘图函数：

```java
private void myDraw () {
    ...
    case GAMEING:
        ...
        if (isBoss == false) {
            //敌机绘制
            for (int i = 0; i < vcEnemy.size(); i++) {
                vcEnemy.elementAt(i).draw(canvas, paint);
            }
        } else {
            //Boss绘制
        }
        break;
    ...
}
```

逻辑函数：

```java
private void myDraw () {
  ...
    //敌机逻辑
    if (isBoss == false) {
```

```
            //敌机逻辑
            for (int i = 0; i < vcEnemy.size(); i++) {
                Enemy en = vcEnemy.elementAt(i);
                //因为容器不断添加敌机 ，那么对敌机 isDead 判定,
                //如果已死亡那么就从容器中删除,对容器起到了优化作用;
                if (en.isDead) {
                    vcEnemy.removeElementAt(i);
                } else {
                    en.logic();
                }
            }
            //生成敌机
            count++;
            if (count % createEnemyTime == 0) {
              for (int i = 0; i < enemyArray[enemyArrayIndex].length; i++) {
                 //苍蝇
                if (enemyArray[enemyArrayIndex][i] == 1) {
                    int x = random.nextInt(screenW - 100) + 50;
                    vcEnemy.addElement(new Enemy(bmpEnemyFly, 1, x, -
                                              50));
                //鸭子左
                } else if (enemyArray[enemyArrayIndex][i] == 2) {
                    int y = random.nextInt(20);
                    vcEnemy.addElement(new Enemy(bmpEnemyDuck, 2, -
                                    50, y));
                //鸭子右
                } else if (enemyArray[enemyArrayIndex][i] == 3) {
                    int y = random.nextInt(20);
                    vcEnemy.addElement(new Enemy(bmpEnemyDuck, 3,
                                              screenW + 50, y));
                }
              }
              //这里判断下一组是否为最后一组(Boss)
              if (enemyArrayIndex == enemyArray.length - 1) {
                  isBoss = true;
              }else{
                  enemyArrayIndex++;
              }
          }
      } else {
            //Boss 逻辑
      }
      ...
}
```

在绘图和逻辑函数中都对以后的 Boss 类留下了接口。

运行项目，效果如图 5-4 所示。

到此敌机与主角都存在了，那么接下来就要开始实现它们之间碰撞的关系。一般飞行射击类型的游戏中，主角与敌机接触后，主角会被扣除血量，而敌机不会；敌机只有在被主角的子弹击中才会扣除血量，当然在后续完善项目中会实现此功能，当前应该实现的是敌机与主角之间的碰撞关系。

修改主角 Player 类，为其添加一个检测主角与敌机碰撞的函数：

```
//判断碰撞(主角与敌机)
public boolean isCollsionWith(Enemy en) {
      int x2 = en.x;
      int y2 = en.y;
      int w2 = en.frameW;
      int h2 = en.frameH;
      if (x >= x2 && x >= x2 + w2) {
            return false;
      } else if (x <= x2 && x + bmpPlayer.getWidth() <= x2) {
            return false;
      } else if (y >= y2 && y >= y2 + h2) {
            return false;
      } else if (y <= y2 && y + bmpPlayer.getHeight() <= y2) {
            return false;
      }
      return true;
}
```

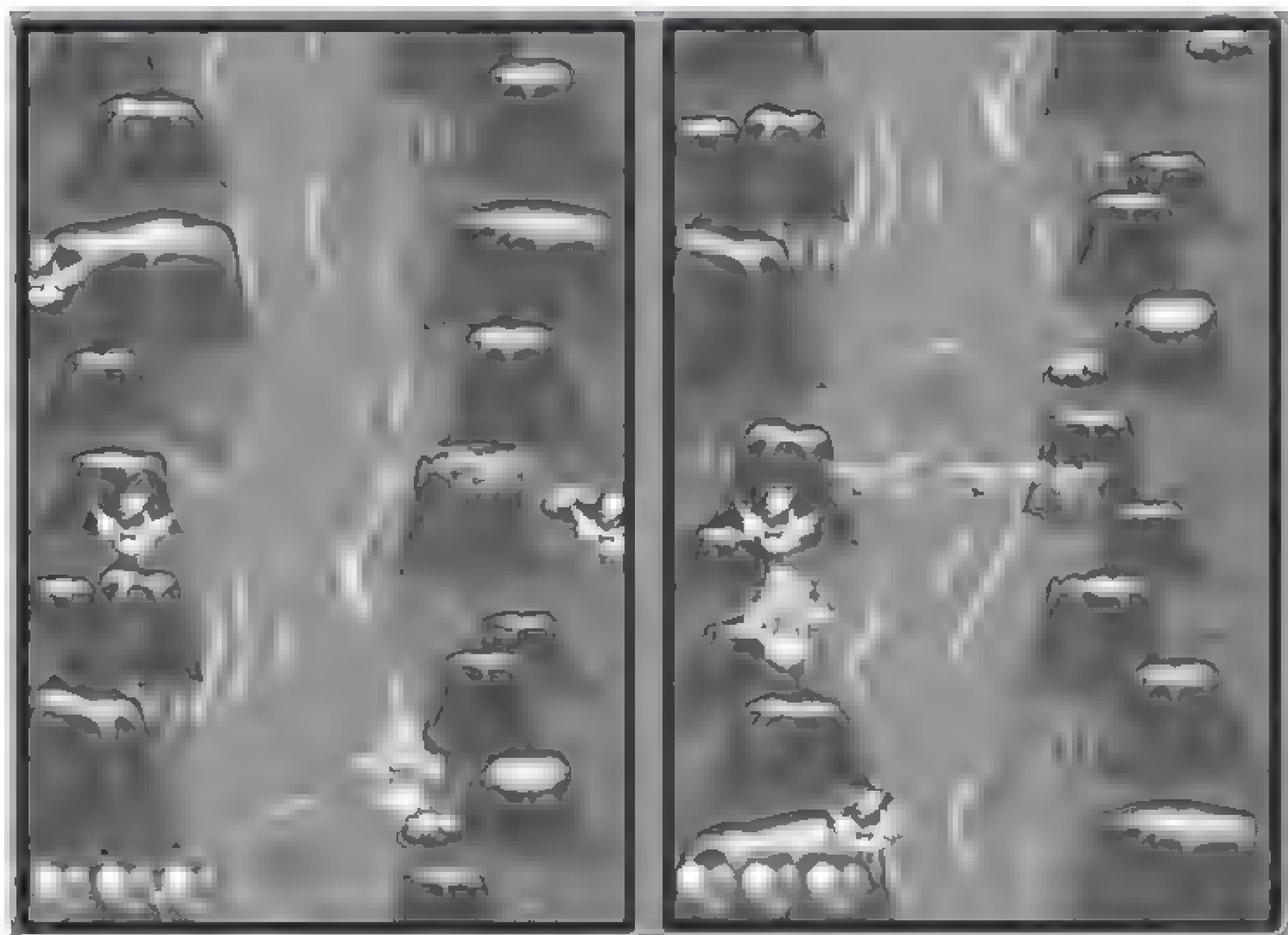


图 5-4　三种运动轨迹类型的敌机

然后修改主视图 MySurfaceView，在游戏中添加处理碰撞逻辑：

```
//处理敌机与主角的碰撞
for (int i = 0; i < vcEnemy.size(); i++) {
      if (player.isCollsionWith(vcEnemy.elementAt(i))) {
            //发生碰撞，主角血量-1
            player.setPlayerHp(player.getPlayerHp() - 1);
      }
}
```

这里还需要再处理一下，当主角与敌机（或者以后主角与敌机的子弹）发生碰撞时，主角极大的可能还处在与敌机碰撞的位置，因为游戏循环逻辑不断循环处理，玩家可能还没来及移动主角，所以增加一个状态，当主角发生碰撞时，设置数秒无敌时间（无敌时间根据需求设置）。

修改 Player 类，添加必须的成员变量：

```
//碰撞后处于无敌时间
//计时器
private int noCollisionCount = 0;
//因为无敌时间
private int noCollisionTime = 60;
//是否碰撞的标识位
private boolean isCollision;
```

修改主角与敌机的碰撞函数：

```
//判断碰撞(主角与敌机)
public boolean isCollsionWith(Enemy en) {
      //是否处于无敌时间
      if (isCollision == false) {
            int x2 = en.x;
            int y2 = en.y;
            int w2 = en.frameW;
            int h2 = en.frameH;
            if (x >= x2 && x >= x2 + w2) {
                  return false;
            } else if (x <= x2 && x + bmpPlayer.getWidth() <= x2) {
                  return false;
            } else if (y >= y2 && y >= y2 + h2) {
                  return false;
            } else if (y <= y2 && y + bmpPlayer.getHeight() <= y2) {
                  return false;
            }
            //碰撞即进入无敌状态
            isCollision = true;
            return true;
      //处于无敌状态，无视碰撞
      } else {
```

```
            return false;
        }
}
```

修改逻辑函数：

```
//主角的逻辑
public void logic() {
        ...
        //处理无敌状态
        if (isCollision) {
                //计时器开始计时
                noCollisionCount++;
                if (noCollisionCount >= noCollisionTime) {
                        //无敌时间过后，接触无敌状态及初始化计数器
                        isCollision = false;
                        noCollisionCount = 0;
                }
        }
        ...
}
```

当主角进入无敌状态时，计时器的递增没有写入碰撞函数，而是写在了主角的逻辑函数中!原因在于，当整个游戏逻辑执行一次时，主角的逻辑也只执行一次；但是主角的碰撞函数并不只是执行一次。如果怪物数量有多个，这个函数将会被调用多次，那么如果将无敌状态的计时器放在主角与敌机的碰撞函数中，计时器每次递增就不是 1 了，而递增的值将与当前存在的敌机数量相同。

虽然此时对主角进入无敌的状态处理完毕，但是对于玩家来说，在视觉上没有任何的变化，那么为了体现主角已进入无敌状态，这里对主角的绘制进行处理，让主角在无敌状态下闪烁。

修改主角的绘制：

```
//主角的绘图函数
public void draw(Canvas canvas, Paint paint) {
        ...
        //绘制主角
        //当处于无敌时间时，让主角闪烁
        if (isCollision) {
                //每 2 次游戏循环，绘制一次主角
                if (noCollisionCount % 2 == 0) {
                        canvas.drawBitmap(bmpPlayer, x, y, paint);
                }
        } else {
                canvas.drawBitmap(bmpPlayer, x, y, paint);
        }
```

```
    ...
}
```

运行项目，效果如图 5-5 所示。

图 5-5 的最右侧图中找不到主角的原因是因为主角与敌机发生碰撞后处于无敌，闪烁的状态，若隐若现。接下来为主角与敌机加上子弹。

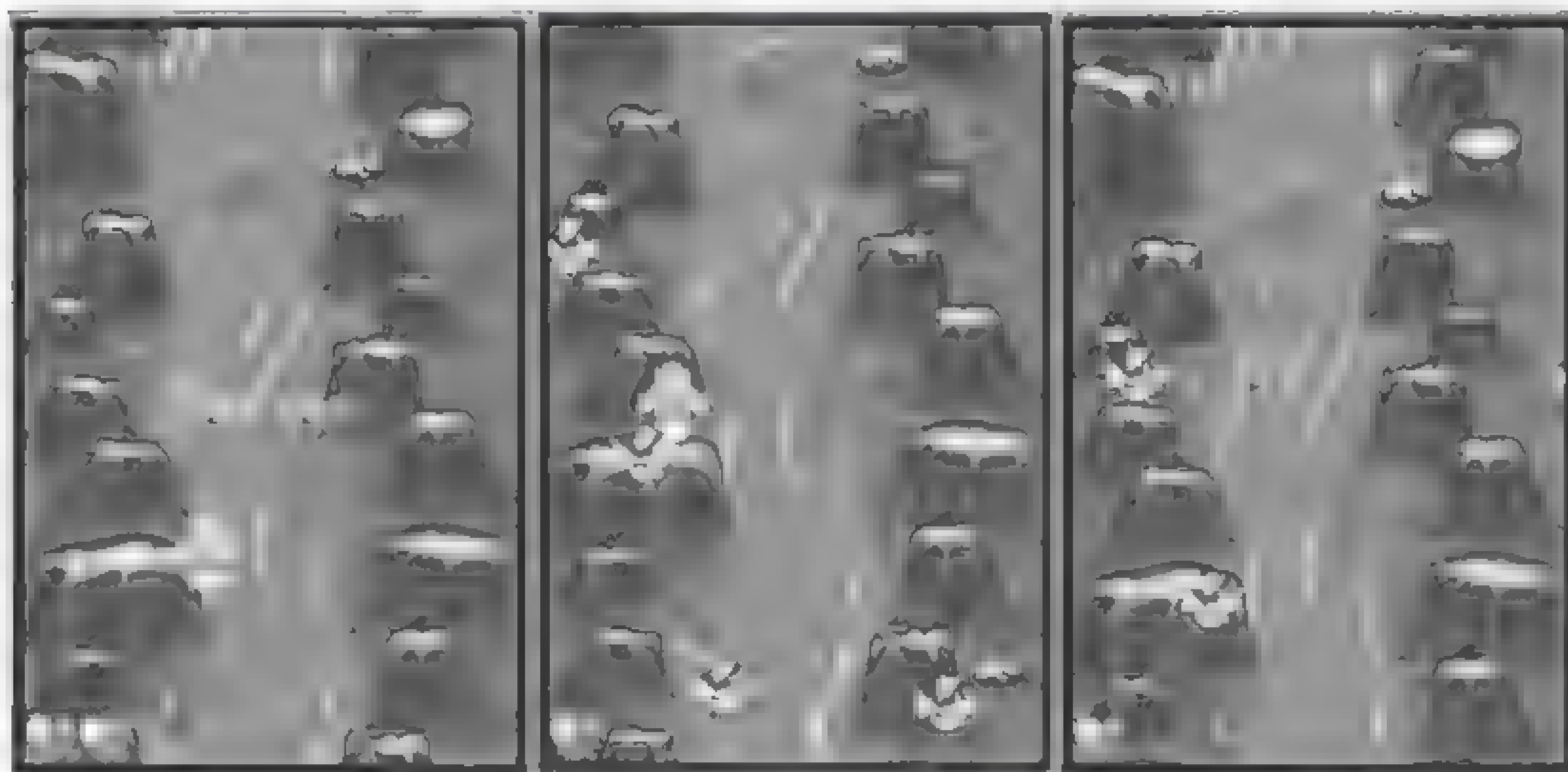


图 5-5　主角与敌机碰撞

首先新建子弹类 Bullet：

```
public class Bullet {
    //子弹图片资源
    public Bitmap bmpBullet;
    //子弹的坐标
    public int bulletX, bulletY;
    //子弹的速度
    public int speed;
    //子弹的种类以及常量
    public int bulletType;
    //主角的
    public static final int BULLET_PLAYER = -1;
    //鸭子的
    public static final int BULLET_DUCK = 1;
    //苍蝇的
    public static final int BULLET_FLY = 2;
    //Boss 的
    public static final int BULLET_BOSS = 3;
    //子弹是否超屏， 优化处理
    public boolean isDead;

    //子弹构造函数
    public Bullet(Bitmap bmpBullet, int bulletX, int bulletY, int
                bulletType) {
        this.bmpBullet = bmpBullet;
```

```
        this.bulletX = bulletX;
        this.bulletY = bulletY;
        this.bulletType = bulletType;
        //不同的子弹类型速度不一
        switch (bulletType) {
        case BULLET_PLAYER:
            speed = 4;
            break;
        case BULLET_DUCK:
            speed = 3;
            break;
        case BULLET_FLY:
            speed = 4;
            break;
        case BULLET_BOSS:
            speed = 5;
            break;
        }
    }
    //子弹的绘制
    public void draw(Canvas canvas, Paint paint) {
        canvas.drawBitmap(bmpBullet, bulletX, bulletY, paint);
    }
    //子弹的逻辑
    public void logic() {
        //不同的子弹类型逻辑不一
        //主角的子弹垂直向上运动
        switch (bulletType) {
        case BULLET_PLAYER:
            bulletY -= speed;
            if (bulletY < -50) {
                isDead = true;
            }
            break;
        //鸭子和苍蝇的子弹都是垂直下落运动
        case BULLET_DUCK:
        case BULLET_FLY:
            bulletY += speed;
            if (bulletY > MySurfaceView.screenH) {
                isDead = true;
            }
            break;
        case BULLET_BOSS:
            //Boss 的子弹逻辑待实现
            break;
        }
```

```
        }
}
```

子弹类设计完成之后（Boss 子弹此时没有添加逻辑处理），在主游戏视图 MySurfaceView 中为敌机与主角添加子弹：

```
//敌机子弹容器
private Vector<Bullet> vcBullet = new Vector<Bullet>();
//添加子弹的计数器
private int countEnemyBullet;
//主角子弹容器
private Vector<Bullet> vcBulletPlayer = new Vector<Bullet>();
//添加子弹的计数器
private int countPlayerBullet;
```

绘图函数：

```
public void myDraw() {
      ...
      case GAMEING:
            ...
            if (isBoss == false) {
                  ...
                  //敌机子弹绘制
                  for (int i = 0; i < vcBullet.size(); i++) {
                        vcBullet.elementAt(i).draw(canvas, paint);
                  }
            } else {
                  //Boss 绘制
            }
            //处理主角子弹绘制
            for (int i = 0; i < vcBulletPlayer.size(); i++) {
                  vcBulletPlayer.elementAt(i).draw(canvas, paint);
            }
            break;
      ...
}
```

逻辑函数：

```
private void logic() {
      ...
      case GAMEING:
            ...
            if (isBoss == false) {
                  ...
                  //每 2 秒添加一个敌机子弹
```

```
            countEnemyBullet++;
            if (countEnemyBullet % 40 == 0) {
                for (int i = 0; i < vcEnemy.size(); i++) {
                    Enemy en = vcEnemy.elementAt(i);
                    //不同类型敌机不同的子弹运行轨迹
                    int bulletType = 0;
                    switch (en.type) {
                    //苍蝇
                    case Enemy.TYPE_FLY:
                        bulletType=Bullet.BULLET_FLY;
                        break;
                    //鸭子
                    case Enemy.TYPE_DUCKL:
                    case Enemy.TYPE_DUCKR:
                        bulletType=Bullet.BULLET_DUCK;
                        break;
                    }
                    vcBullet.add(new Bullet (bmpEnemyBullet,
              en.x + 10, en.y + 20, bulletType));
                }
            }
            //处理敌机子弹逻辑
            for (int i = 0; i < vcBullet.size(); i++) {
                Bullet b = vcBullet.elementAt(i);
                if (b.isDead) {
                    vcBullet.removeElement(b);
                } else {
                    b.logic();
                }
            }
        } else {
            //Boss 逻辑
        }
        //每1秒添加一个主角子弹
        countPlayerBullet++;
        if (countPlayerBullet % 20 == 0) {
            vcBulletPlayer.add(new Bullet(bmpBullet, player.x + 15,
                 player.y - 20, Bullet.BULLET_PLAYER));
        }
        //处理主角子弹逻辑
        for (int i = 0; i < vcBulletPlayer.size(); i++) {
            Bullet b = vcBulletPlayer.elementAt(i);
            if (b.isDead) {
                vcBulletPlayer.removeElement(b);
            } else {
                b.logic();
```

```
                    }
                }
                break;
        ...
}
```

敌机与敌机子弹的绘制与逻辑都写在了 Boss 没有出现的状态中，因为当 Boss 出现时，敌机与敌机子弹都将消失；而主角与主角子弹在 Boss 出现时也会存在，所以主角以及主角子弹不受 Boss 影响.

运行项目，效果如图 5-6 所示。

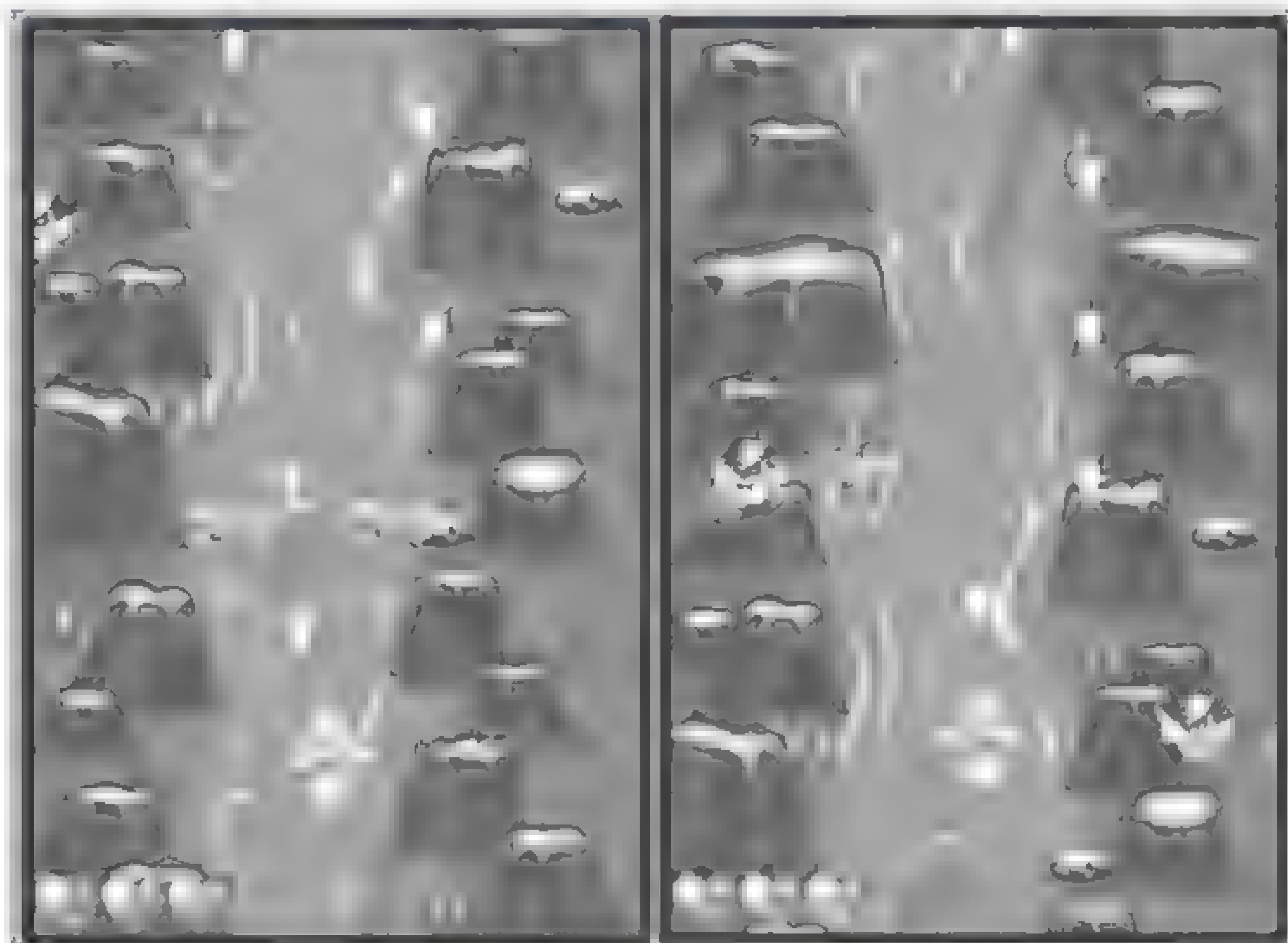


图 5-6　主角与敌机的子弹

子弹的绘制与逻辑除 Boss 外都已完成，紧跟着就应该添加敌机与主角子弹之间的碰撞逻辑。

分析一下：当主角的子弹与敌机碰撞后，敌机死亡消失，并且产生爆炸效果；而敌机的子弹与主角碰撞后，主角的血量减 1，暂且不实现爆炸效果。

首先修改 Player 类，为主角添加与敌机子弹碰撞的逻辑函数：

```
//判断碰撞(主角与敌机子弹)
public boolean isCollsionWith(Bullet bullet) {
      //是否处于无敌时间
      if (isCollision == false) {
            int x2 = bullet.bulletX;
            int y2 = bullet.bulletY;
            int w2 = bullet.bmpBullet.getWidth();
            int h2 = bullet.bmpBullet.getHeight();
            if (x >= x2 && x >= x2 + w2) {
                  return false;
```

```
            } else if (x <= x2 && x + bmpPlayer.getWidth() <= x2) {
                return false;
            } else if (y >= y2 && y >= y2 + h2) {
                return false;
            } else if (y <= y2 && y + bmpPlayer.getHeight() <= y2) {
                return false;
            }
            //碰撞即进入无敌状态
            isCollision = true;
            return true;
            //处于无敌状态，无视碰撞
        } else {
            return false;
        }
}
```

此方法与主角跟敌机碰撞函数几乎一样，不再赘述；在主视图 MySurfaceView 的逻辑函数中添加以下代码：

```
//处理敌机子弹与主角碰撞
for (int i = 0; i < vcBullet.size(); i++) {
        if (player.isCollsionWith(vcBullet.elementAt(i))) {
            //发生碰撞，主角血量-1
            player.setPlayerHp(player.getPlayerHp() - 1);
            //当主角血量小于 0，判定游戏失败
            if (player.getPlayerHp() <= -1) {
                gameState = GAME_LOST;
            }
        }
}
```

然后添加主角子弹与敌机的碰撞，修改 Enemy 类，添加碰撞函数：

```
//判断碰撞(敌机与主角子弹碰撞)
public boolean isCollsionWith(Bullet bullet) {
        int x2 = bullet.bulletX;
        int y2 = bullet.bulletY;
        int w2 = bullet.bmpBullet.getWidth();
        int h2 = bullet.bmpBullet.getHeight();
        if (x >= x2 && x >= x2 + w2) {
            return false;
        } else if (x <= x2 && x + frameW <= x2) {
            return false;
        } else if (y >= y2 && y >= y2 + h2) {
            return false;
        } else if (y <= y2 && y + frameH <= y2) {
```

```
            return false;
        }
        //发生碰撞，让其死亡
        isDead = true;
        return true;
}
```

在主视图 MySurfaceView 中添加逻辑：

```
//处理主角子弹与敌机碰撞
for (int i = 0; i < vcBulletPlayer.size(); i++) {
        //取出主角子弹容器的每个元素
        Bullet blPlayer = vcBulletPlayer.elementAt(i);
        for (int j = 0; j < vcEnemy.size(); j++) {
            //取出敌机容器的每个元与主角子弹遍历判断
            if(vcEnemy.elementAt(j).isCollsionWith(blPlayer)){
                //添加爆炸效果
            }
        }
}
```

这里留下了爆炸的接口，当主角子弹与敌机发生碰撞后，将在爆炸效果的容器中添加一个爆炸类对象实例。下面就来实现爆炸类，以及爆炸的绘制与逻辑。

新建类 Boom：

```
public class Boom {
        //爆炸效果资源图
        private Bitmap bmpBoom;
        //爆炸效果的位置坐标
        private int boomX, boomY;
        //爆炸动画播放当前的帧下标
        private int cureentFrameIndex;
        //爆炸效果的总帧数
        private int totleFrame;
        //每帧的宽高
        private int frameW, frameH;
        //是否播放完毕，优化处理
        public boolean playEnd;
        //爆炸效果的构造函数
        public Boom(Bitmap bmpBoom, int x, int y, int totleFrame) {
            this.bmpBoom = bmpBoom;
            this.boomX = x;
            this.boomY = y;
            this.totleFrame = totleFrame;
            frameW = bmpBoom.getWidth() / totleFrame;
            frameH = bmpBoom.getHeight();
```

```
    }
    //爆炸效果绘制
    public void draw(Canvas canvas, Paint paint) {
        canvas.save();
        canvas.clipRect(boomX, boomY, boomX + frameW, boomY + frameH);
        canvas.drawBitmap(bmpBoom, boomX - cureentFrameIndex * frameW,
boomY, paint);
        canvas.restore();
    }
    //爆炸效果的逻辑
    public void logic() {
        if (cureentFrameIndex < totleFrame) {
            cureentFrameIndex++;
        } else {
            playEnd = true;
        }
    }
}
```

爆炸效果分两种类型，一种是普通敌机的，一种是 Boss 的爆炸效果，当前留下 Boss 爆炸效果的逻辑处理接口，暂且不实现。

在主视图 MySurfaceView 中添加爆炸效果，代码如下：

```
//爆炸效果容器
private Vector<Boom> vcBoom = new Vector<Boom>();
```

绘图函数：

```
//爆炸效果绘制
for (int i = 0; i < vcBoom.size(); i++) {
    vcBoom.elementAt(i).draw(canvas, paint);
}
```

逻辑函数：

```
//处理主角子弹与敌机碰撞
for (int i = 0; i < vcBulletPlayer.size(); i++) {
    //取出主角子弹容器的每个元素
    Bullet blPlayer = vcBulletPlayer.elementAt(i);
    for (int j = 0; j < vcEnemy.size(); j++) {
        //添加爆炸效果
        //取出敌机容器的每个元与主角子弹遍历判断
        if (vcEnemy.elementAt(j).isCollsionWith(blPlayer)) {
            vcBoom.add(new Boom(bmpBoom, vcEnemy.elementAt(j).x,
                              vcEnemy.elementAt(j).y, 7));
        }
```

```
        }
}
//爆炸效果逻辑
for (int i = 0; i < vcBoom.size(); i++) {
        Boom boom = vcBoom.elementAt(i);
        if (boom.playEnd) {
                //播放完毕的从容器中删除
                vcBoom.removeElementAt(i);
        } else {
                vcBoom.elementAt(i).logic();
        }
}
```

运行项目，效果如图 5-7 所示。

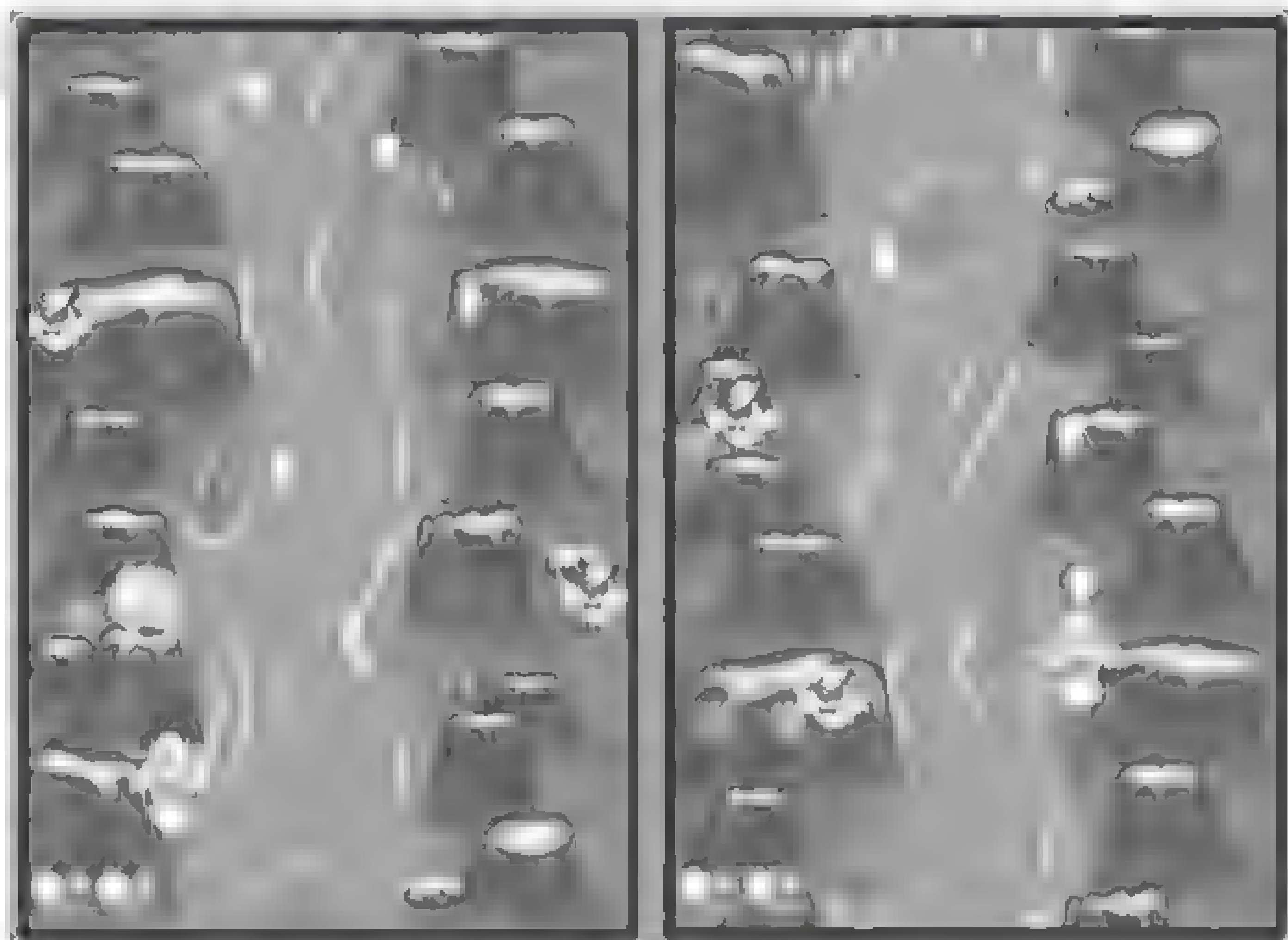


图 5-7　爆炸效果

最后就是设计最终 Boss。既然是 Boss，那么就不能被主角一颗子弹打死，所以血量肯定要多，并且子弹逻辑除了垂直运动外，还为 Boss 添加一个疯狂的状态。在疯狂状态下，Boss 会同时发射八方向子弹，使整个游戏困难度提高。

也就是说 Boss 拥有两套子弹运动轨迹，创建 Boss 垂直下落的子弹时直接使用之前敌机的子弹轨迹即可。然后再为 Bullet 子弹类添加 Boss 同时发射八方向子弹时运动轨迹和逻辑。

Bullet 类修改如下：

```
//Boss 疯狂状态下子弹相关成员变量
private int dir;//当前 Boss 子弹方向
//8 方向常量
public static final int DIR_UP = -1;
public static final int DIR_DOWN = 2;
public static final int DIR_LEFT = 3;
```

```
public static final int DIR_RIGHT = 4;
public static final int DIR_UP_LEFT = 5;
public static final int DIR_UP_RIGHT = 6;
public static final int DIR_DOWN_LEFT = 7;
public static final int DIR_DOWN_RIGHT = 8;
```

再添加一个子弹类的构造函数：

```
/**
 * 专用于处理Boss疯狂状态下创建的子弹
 * @param bmpBullet
 * @param bulletX
 * @param bulletY
 * @param bulletType
 * @param Dir
 */
public Bullet(Bitmap bmpBullet, int bulletX, int bulletY, int
                    bulletType, int dir) {
      this.bmpBullet = bmpBullet;
      this.bulletX = bulletX;
      this.bulletY = bulletY;
      this.bulletType = bulletType;
      speed = 5;
      this.dir = dir;
}
```

子弹逻辑中Boss类型子弹的处理：

```
//子弹的逻辑
public void logic() {
      …
      //Boss疯狂状态下的8方向子弹逻辑
      case BULLET_BOSS:
            //Boss疯狂状态下的子弹逻辑待实现
            switch (dir) {
            //方向上的子弹
            case DIR_UP:
                  bulletY -= speed;
                  break;
            //方向下的子弹
            case DIR_DOWN:
                  bulletY += speed;
                  break;
            //方向左的子弹
            case DIR_LEFT:
                  bulletX -= speed;
```

```
                break;
            //方向右的子弹
            case DIR_RIGHT:
                bulletX += speed;
                break;
            //方向左上的子弹
            case DIR_UP_LEFT:
                bulletY -= speed;
                bulletX -= speed;
                break;
            //方向右上的子弹
            case DIR_UP_RIGHT:
                bulletX += speed;
                bulletY -= speed;
                break;
            //方向左下的子弹
            case DIR_DOWN_LEFT:
                bulletX -= speed;
                bulletY += speed;
                break;
            //方向右下的子弹
            case DIR_DOWN_RIGHT:
                bulletY += speed;
                bulletX += speed;
                break;
            }
            //边界处理
            if (bulletY > MySurfaceView.screenH || bulletY <= -40 ||
                    bulletX > MySurfaceView.screenW || bulletX <= -40) {
                isDead = true;
            }
            break;
        ...
}
```

封装Boss类，新建类“Boss”：

```
public class Boss {
    //Boss的血量
    public int hp = 50;
    //Boss的图片资源
    private Bitmap bmpBoss;
    //Boss坐标
    public int x, y;
    //Boss每帧的宽高
    public int frameW, frameH;
    //Boss当前帧下标
```

```
    private int frameIndex;
    //Boss 运动的速度
    private int speed = 5;
    //Boss 的运动轨迹
    //一定时间会向着屏幕下方运动，并且发射大范围子弹，（是否狂态）
    //正常状态下 ，子弹垂直朝下运动
    private boolean isCrazy;
    //进入疯狂状态的状态时间间隔
    private int crazyTime = 200;
    //计数器
    private int count;

    //Boss 的构造函数
    public Boss(Bitmap bmpBoss) {
        this.bmpBoss = bmpBoss;
        frameW = bmpBoss.getWidth() / 10;
        frameH = bmpBoss.getHeight();
        //Boss 的 X 坐标居中
        x = MySurfaceView.screenW / 2 - frameW / 2;
        y = 0;
    }
    //Boss 的绘制
    public void draw(Canvas canvas, Paint paint) {
        canvas.save();
        canvas.clipRect(x, y, x + frameW, y + frameH);
        canvas.drawBitmap(bmpBoss, x - frameIndex * frameW, y, paint);
        canvas.restore();
    }

    //Boss 的逻辑
    public void logic() {
        //不断循环播放帧形成动画
        frameIndex++;
        if (frameIndex >= 10) {
            frameIndex = 0;
        }
        //没有疯狂的状态
        if (isCrazy == false) {
            x += speed;
            if (x + frameW >= MySurfaceView.screenW) {
                speed = -speed;
            } else if (x <= 0) {
                speed = -speed;
            }
            count++;
            if (count % crazyTime == 0) {
```

```
                    isCrazy = true;
                    speed = 24;
                }
                //疯狂的状态
            } else {
                speed -= 1;
                //当Boss返回时创建大量子弹
                if (speed == 0) {
                    //添加8方向子弹
                    MySurfaceView.vcBulletBoss.add(new
Bullet(MySurfaceView.bmpBossBullet, x+30, y, Bullet.BULLET_BOSS,
Bullet.DIR_UP));
                    MySurfaceView.vcBulletBoss.add(new
Bullet(MySurfaceView.bmpBossBullet, x+30, y, Bullet.BULLET_BOSS,
Bullet.DIR_DOWN));
                    MySurfaceView.vcBulletBoss.add(new
Bullet(MySurfaceView.bmpBossBullet, x+30, y, Bullet.BULLET_BOSS,
Bullet.DIR_LEFT));
                    MySurfaceView.vcBulletBoss.add(new
Bullet(MySurfaceView.bmpBossBullet, x+30, y, Bullet.BULLET_BOSS,
Bullet.DIR_RIGHT));
                    MySurfaceView.vcBulletBoss.add(new
Bullet(MySurfaceView.bmpBossBullet, x+30, y, Bullet.BULLET_BOSS,
Bullet.DIR_UP_LEFT));
                    MySurfaceView.vcBulletBoss.add(new
Bullet(MySurfaceView.bmpBossBullet, x+30, y, Bullet.BULLET_BOSS,
Bullet.DIR_UP_RIGHT));
                    MySurfaceView.vcBulletBoss.add(new
Bullet(MySurfaceView.bmpBossBullet, x+30, y, Bullet.BULLET_BOSS,
Bullet.DIR_DOWN_LEFT));
                    MySurfaceView.vcBulletBoss.add(new
Bullet(MySurfaceView.bmpBossBullet, x+30, y, Bullet.BULLET_BOSS,
Bullet.DIR_DOWN_RIGHT));
                }
                y += speed;
                if (y <= 0) {
                    //恢复正常状态
                    isCrazy = false;
                    speed = 5;
                }
            }
        }

        //判断碰撞(Boss被主角子弹击中)
        public boolean isCollsionWith(Bullet bullet) {
            int x2 = bullet.bulletX;
```

```
            int y2 = bullet.bulletY;
            int w2 = bullet.bmpBullet.getWidth();
            int h2 = bullet.bmpBullet.getHeight();
            if (x >= x2 && x >= x2 + w2) {
                  return false;
            } else if (x <= x2 && x + frameW <= x2) {
                  return false;
            } else if (y >= y2 && y >= y2 + h2) {
                  return false;
            } else if (y <= y2 && y + frameH <= y2) {
                  return false;
            }
            return true;
      }

      //设置 Boss 血量
      public void setHp(int hp) {
            this.hp = hp;
      }
}
```

Boss 类与 Boss 子弹都添加完毕后，在主视图 MySurfaceView 中添加 Boss 以及 Boss 子弹的绘制与逻辑处理：

```
      //声明 Boss
      private Boss boss;
      //Boss 的子弹容器
      public static Vector<Bullet> vcBulletBoss;
```

游戏初始化函数：

```
private void initGame() {
                  //...
                  //实例 boss 对象
                  boss = new Boss(bmpEnemyBoos);
                  //实例 Boss 子弹容器
                  vcBulletBoss = new Vector<Bullet>();
            //...
}
```

绘制函数：

```
public void myDraw() {
      ...
```

```
        //Boos 的绘制
        boss.draw(canvas, paint);
        //Boss 子弹逻辑
        for (int i = 0; i < vcBulletBoss.size(); i++) {
                vcBulletBoss.elementAt(i).draw(canvas, paint);
        }
        ...
}
```

逻辑函数：

```
private void logic() {
        ...
        //Boss 相关逻辑
        //每 0.5 秒添加一个主角子弹
        boss.logic();
        if (countPlayerBullet % 10 == 0) {
                //Boss 的没发疯之前的普通子弹
                vcBulletBoss.add(new Bullet(bmpBossBullet, boss.x + 35,
                                                boss.y + 40, Bullet.BULLET_FLY));
        }
        //Boss 子弹逻辑
        for (int i = 0; i < vcBulletBoss.size(); i++) {
                Bullet b = vcBulletBoss.elementAt(i);
                if (b.isDead) {
                        vcBulletBoss.removeElement(b);
                } else {
                        b.logic();
                }
        }
        //Boss 子弹与主角的碰撞
        for (int i = 0; i < vcBulletBoss.size(); i++) {
                if (player.isCollsionWith(vcBulletBoss.elementAt(i))) {
                        //发生碰撞，主角血量-1
                        player.setPlayerHp(player.getPlayerHp() - 1);
                        //当主角血量小于 0，判定游戏失败
                        if (player.getPlayerHp() <= -1) {
                                        gameState = GAME_LOST;
                        }
                }
        }
        //Boss 被主角子弹击中，产生爆炸效果
        for (int i = 0; i < vcBulletPlayer.size(); i++) {
                Bullet b = vcBulletPlayer.elementAt(i);
                if (boss.isCollsionWith(b)) {
                        if (boss.hp <= 0) {
```

```
                    //游戏胜利
                    gameState = GAME_WIN;
                } else {
                    //及时删除本次碰撞的子弹，防止重复判定此子弹与Boss碰撞
                    b.isDead = true;
                    //Boss血量减1
                    boss.setHp(boss.hp - 1);
                    //在Boss上添加三个Boss爆炸效果
                    vcBoom.add(new Boom(bmpBoosBoom, boss.x + 25,
                                boss.y + 30, 5));
                    vcBoom.add(new Boom(bmpBoosBoom, boss.x + 35,
                               boss.y + 40, 5));
                    vcBoom.add(new Boom(bmpBoosBoom, boss.x + 45,
                               boss.y + 50, 5));


                }
            }
        }
        ...
}
```

运行项目，效果如图5-8所示。

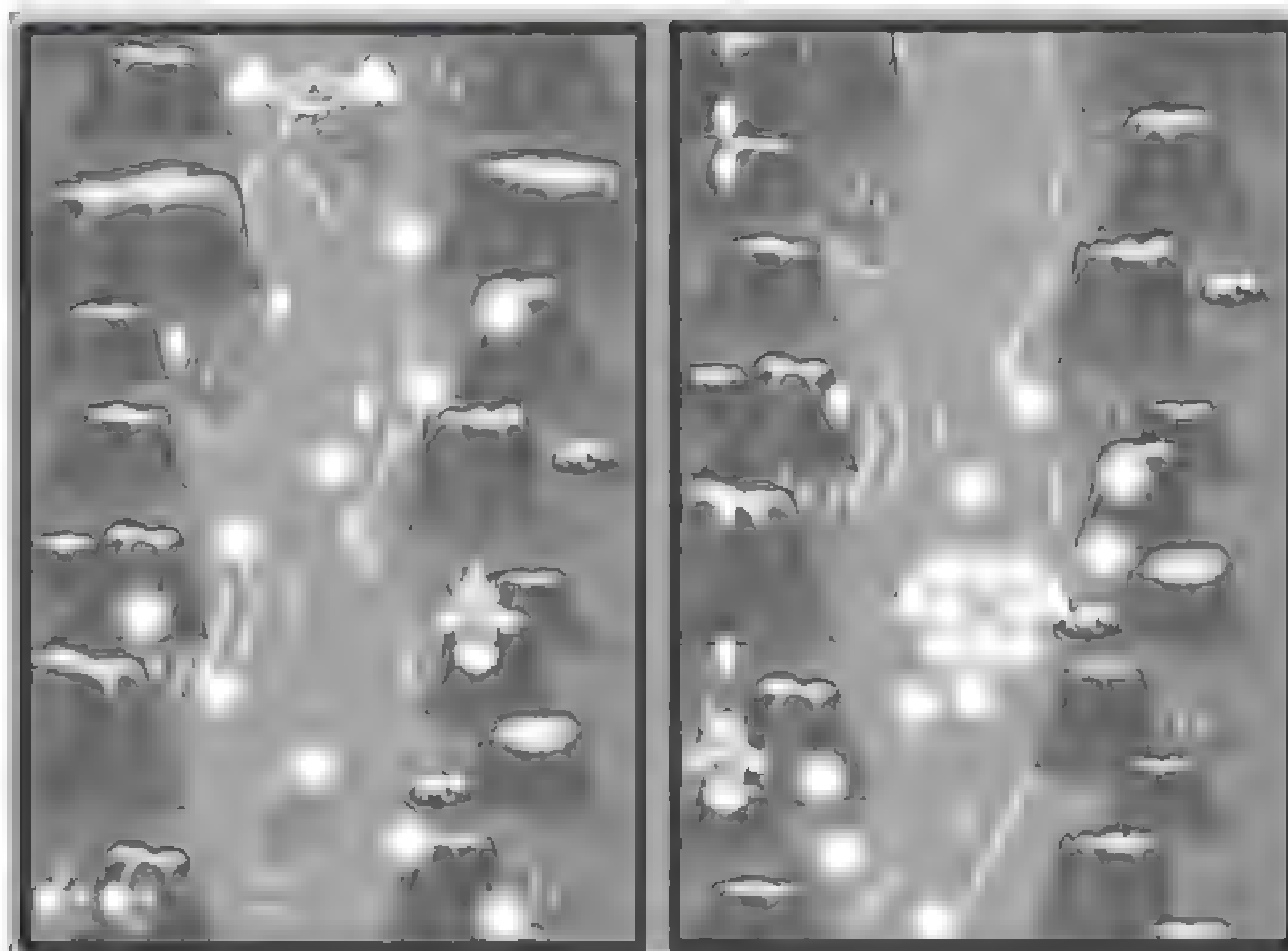


图5-8　Boss与Boss子弹

5.5 游戏胜利与结束界面

对于游戏胜利与失败的条件，在 5.4 小节中都已经有了判定，这里只需要做的是在游戏胜利或失败的状态下绘制相对应的背景图即可。

在主视图 MySurfaceView 的绘图函数中，添加胜利和失败游戏状态的绘制代码：

```
/**
 * 游戏绘图
 */
public void myDraw() {
        ...
        case GAME_WIN:
                canvas.drawBitmap(bmpGameWin, 0, 0, paint);
                break;
        case GAME_LOST:
                canvas.drawBitmap(bmpGameLost, 0, 0, paint);
        ...
}
```

胜利和失败的界面如图 5-9 所示。



图 5-9　胜利和失败的界面

5.6 游戏细节处理

5.6.1 游戏 Back 返回键处理

介绍 SurfaceView 机制时，讲述过当用户单击 Back 返回按键时，当前运行的应用会被切入后台，并且重新进入应用时，会重启活动。而且当设备内存吃紧时，切入后台的程序随时都有可能被切断；所以在游戏开发中，一般都会屏蔽或者处理掉此按钮。

针对当前飞行射击的游戏来说，当游戏处于进行状态，或者游戏胜利与失败状态时，单击返回按键返回菜单即可，其他游戏状态下不进行任何的处理，放置程序切入后台。

在游戏中只需要将此函数设置在主视图 MySurfaceView 的构造函数中即可：

```
public boolean onKeyDown(int keyCode, KeyEvent event) {
    //处理 back 返回按键
    if (keyCode == KeyEvent.KEYCODE_BACK) {
        //游戏胜利、失败、进行时都默认返回菜单
        if (gameState == GAMEING || gameState == GAME_WIN ||
gameState == GAME_LOST) {
            gameState = GAME_MENU;
            isBoss=false;
            //重置游戏
            initGame();
            //重置怪物出场
            enemyArrayIndex = 0;
        }else if(gameState == GAME_MENU){
            //当前游戏状态在菜单界面，默认返回按键退出游戏
            MainActivity.instance.finish();
            System.exit(0);
        }
        //表示此按键已处理，不再交给系统处理，
        //从而避免游戏被切入后台
        return true;
    }
    ...
}
```

这里需要注意，虽然处理了 Back 按键，但发现是没有效果的。如果既想处理 Back 事件，又不让系统将应用切入后台，除了获取 Back 按键事件后利用 return true 处理外，还要让当前视图（游戏主视图）设置以下函数：

```
setFocusableInTouchMode(true);
```

5.6.2　为游戏设置背景常亮

一般手机应用运行时，在用户没有任何操作的情况下，屏幕在一段时间后默认进入省电（屏幕变暗）、锁屏模式，那么如果想让游戏保持背景常亮的话，只需要调用下面函数即可：

M view.View.setKeepScreenOn(boolean keepScreenOn)

作用：保持背景灯光常亮

参数：true 表示设置常亮，false 表示不保持。默认不保持

5.7　本章小结

本章介绍了一个游戏的整体开发流程，以及对自定义类的封装和一些开发过程中应该注意的细节处理；当然在此项目中只做了一个关卡，一些功能没有实现，比如游戏中暂停的状态、游戏的帮助说明、在主菜单添加退出按钮、游戏的背景音乐、游戏的存储等等。

但是这些没有完成的部分并不影响什么，本章节重点还是让大家在脑中形成一个大体的游戏框架，只要基础扎实，在此基础上扩展添加功能就轻而易举了；有兴趣的可以为此项目去修改和添加新功能，让游戏更加丰富完整。

第 6 章

游戏开发提高篇

从本章节可以学习到：

- 360°平滑游戏导航摇杆
- 多触点实现图片缩放
- 触屏手势识别
- 加速度传感器
- 9patch 工具的使用
- 代码实现截屏功能
- 效率检视工具
- 游戏视图与系统组件共同显示
- 蓝牙对战游戏
- 网络游戏开发基础
- 本地化与国际化



6.1 360° 平滑游戏导航摇杆

在 Android 系统中很多机型是没有实体导航按键的，那么如果想让一个游戏在所有 Android 系统的机型上运行，就要利用 Android 系统都支持触屏的特点来进行设计。既然所有 Android 系统都支持触屏，那么就可以想到，在屏幕上绘制一个游戏摇杆供用户操作游戏，这也是目前 Android 游戏开发中最常用的一种做法了。

下面就来实现 Android 手机上的 360° 平滑游戏摇杆吧！首先观察如图 6-1 所示的效果。

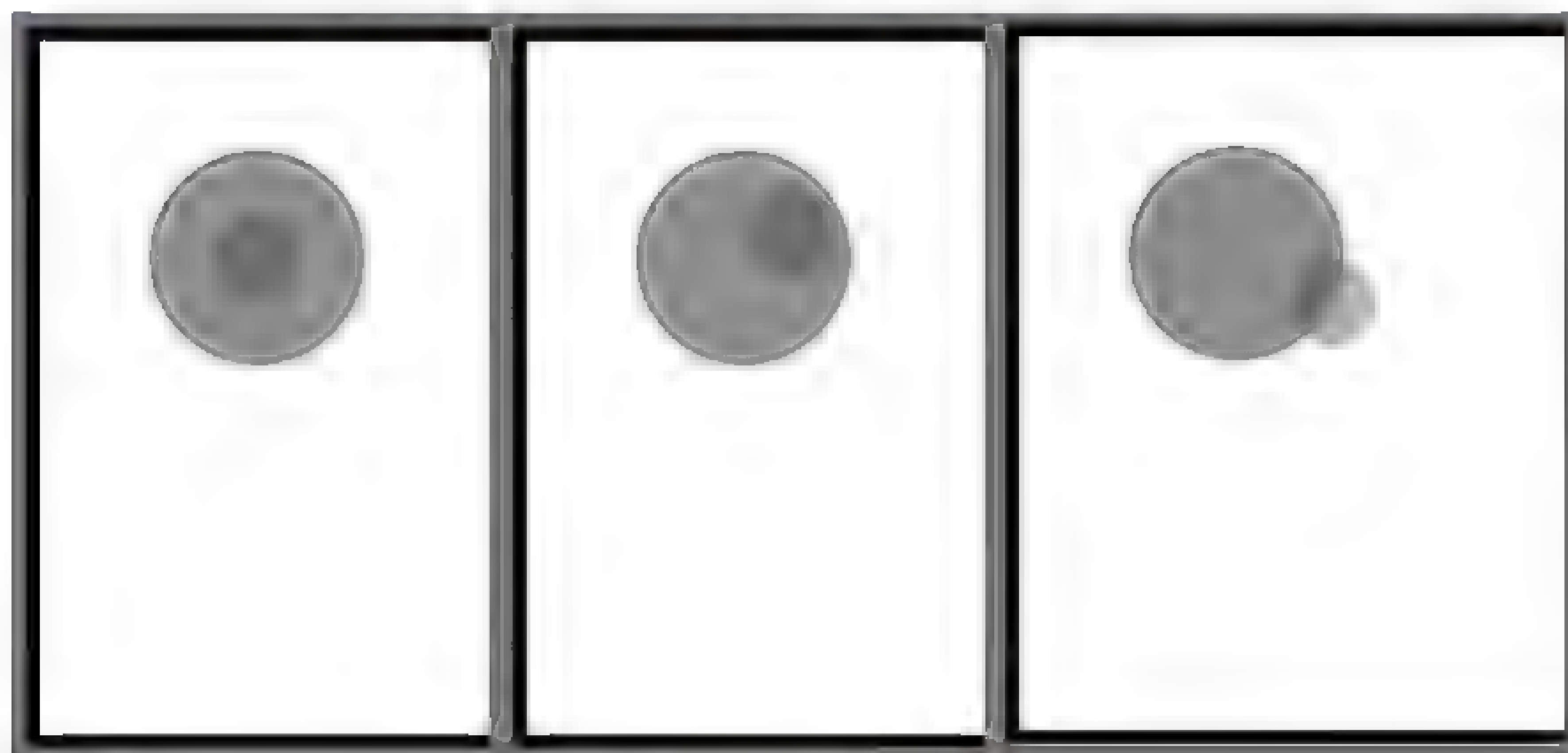


图 6-1 摇杆示意图

图 6-1 是一个摇杆的示意图，从图中加以分析：

- 玩家操作的应该是中心红色的小圆；
- 小圆的最大活动范围是围绕大圆做圆周运动；
- 既然小圆有活动范围，那么当用户的触屏点在大圆以外的位置，那么小圆的角度应该与用户触点的角度相同。

首先实现的应该是在屏幕上绘制两个大小不一的圆形，并且让小圆中心点围绕大圆做圆周运动。

新建项目“RockerProject”，游戏框架为 SurfaceView 游戏框架，项目对应的源代码为“6-1（360° 平滑游戏摇杆）”。

修改 MySurfaceView：

```
//定义两个圆形的中心点坐标与半径
private float smallCenterX = 120, smallCenterY = 120,
                              smallCenterR = 20;
private float BigCenterX = 120, BigCenterY = 120,
```

```
                                BigCenterR = 40;
//当前圆周运动的角度
private int angle;

//修改绘图函数:
public void myDraw() {
	...
	//绘制大圆
	paint.setAlpha(0x77);
	canvas.drawCircle(BigCenterX, BigCenterY, BigCenterR, paint);
	//绘制小圆
	canvas.drawCircle(smallCenterX, smallCenterY, smallCenterR, paint);
	...
}
```

新封装一个圆周运动时，得到小圆坐标的方法：

```
/**
 * 小圆针对于大圆做圆周运动时，设置小圆中心点的坐标位置
 * @param centerX
 *            围绕的圆形(大圆)中心点X坐标
 * @param centerY
 *            围绕的圆形(大圆)中心点Y坐标
 * @param R
 *                      围绕的圆形(大圆)半径
 * @param rad
 *            旋转的弧度
 */
public void setSmallCircleXY(float centerX, float centerY, float R,
double rad) {
	//获取圆周运动的X坐标
	smallCenterX = (float) (R * Math.cos(rad)) + centerX;
	//获取圆周运动的Y坐标
	smallCenterY = (float) (R * Math.sin(rad)) + centerY;
}
```

这里是根据角度弧度的转换，再通过三角函数定理得到小圆坐标位置的。

逻辑函数：

```
private void logic() {
	//让角度在0~360循环
	angle++;
	if (angle >= 360) {
		angle = 0;
	}
	//弧度 = 角度 PI/*180
```

```
        setSmallCircleXY(BigCenterX, BigCenterY, BigCenterR,
angle * Math.PI / 180);
}
```

运行项目，效果如图 6-2 所示。

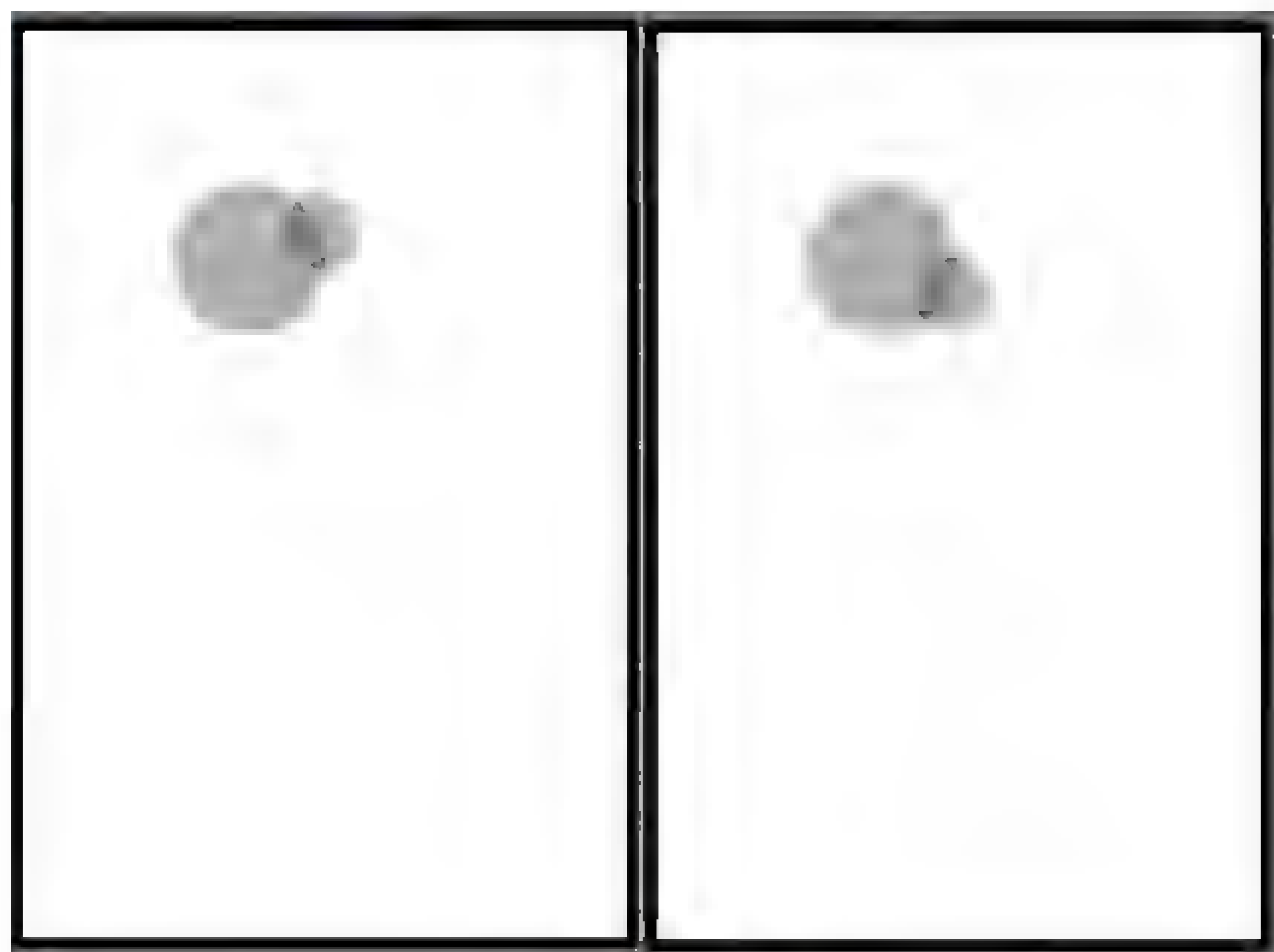


图 6-2　圆周运动

此步完成之后，下面就应该考虑用户触点的位置，大概分为两种情况：

- 用户触点位置在大圆内或者大圆上，那么小圆的中心点直接跟随玩家触点位置即可；
- 用户触点位置在大圆外，那么小圆中心肯定在大圆的圆周上，但是小圆所在大圆上的角度，应该等同于用户触点位置相对于大圆的角度。

首先删去刚才在逻辑函数中的代码，然后封装一个得到玩家触点相对于大圆角度的方法：

```
/**
 * 得到两点之间的弧度
 * @param px1    第一个点的 X 坐标
 * @param py1    第一个点的 Y 坐标
 * @param px2    第二个点的 X 坐标
 * @param py2    第二个点的 Y 坐标
 * @return
 */
public double getRad(float px1, float py1, float px2, float py2) {
      //得到两点 X 的距离
      float x = px2 - px1;
      //得到两点 Y 的距离
      float y = py1 - py2;
      //算出斜边长
      float Hypotenuse = (float) Math.sqrt(Math.pow(x, 2) + Math.pow(y,2));
```

```
        //得到这个角度的余弦值（通过三角函数中的定理 ：邻边/斜边=角度余弦值）
        float cosAngle = x / Hypotenuse;
        //通过反余弦定理获取其角度的弧度
        float rad = (float) Math.acos(cosAngle);
        //当触屏的位置Y坐标<摇杆的Y坐标，取反值-0~-180
        if (py2 < py1) {
            rad = -rad;
        }
        return rad;
}
```

修改触屏监听函数：

```
public boolean onTouchEvent(MotionEvent event) {
        //当用户手指抬起，应该恢复小圆到初始位置
        if (event.getAction() == MotionEvent.ACTION_UP) {
            smallCenterX = BigCenterX;
            smallCenterY = BigCenterY;
        } else {
            int pointX = (int) event.getX();
            int pointY = (int) event.getY();
            //判断用户点击的位置是否在大圆内
            if (Math.sqrt(Math.pow((BigCenterX - (int) event.getX()), 2) +
Math.pow((BigCenterY - (int) event.getY()), 2)) <= BigCenterR) {
                //让小圆跟随用户触点位置移动
                smallCenterX = pointX;
                smallCenterY = pointY;
            } else {
                setSmallCircleXY(BigCenterX, BigCenterY, BigCenterR,
getRad(BigCenterX, BigCenterY, pointX, pointY));
            }
        }
        return true;
}
```

运行项目，效果如图6-3所示。

到此整个摇杆的制作就完成了，那么如何在游戏中使用它呢？其实很简单。

假如需要使用摇杆控制游戏主角的移动，那么首先将整个360° 分成4等分或者8等分，对应主角的4方向或者8方向；然后通过封装的两点之间得到弧度的函数获取摇杆弧度，将其弧度转换成角度，再将摇杆的角度与之前的360° 分成的4或8等分范围比对处理即可。

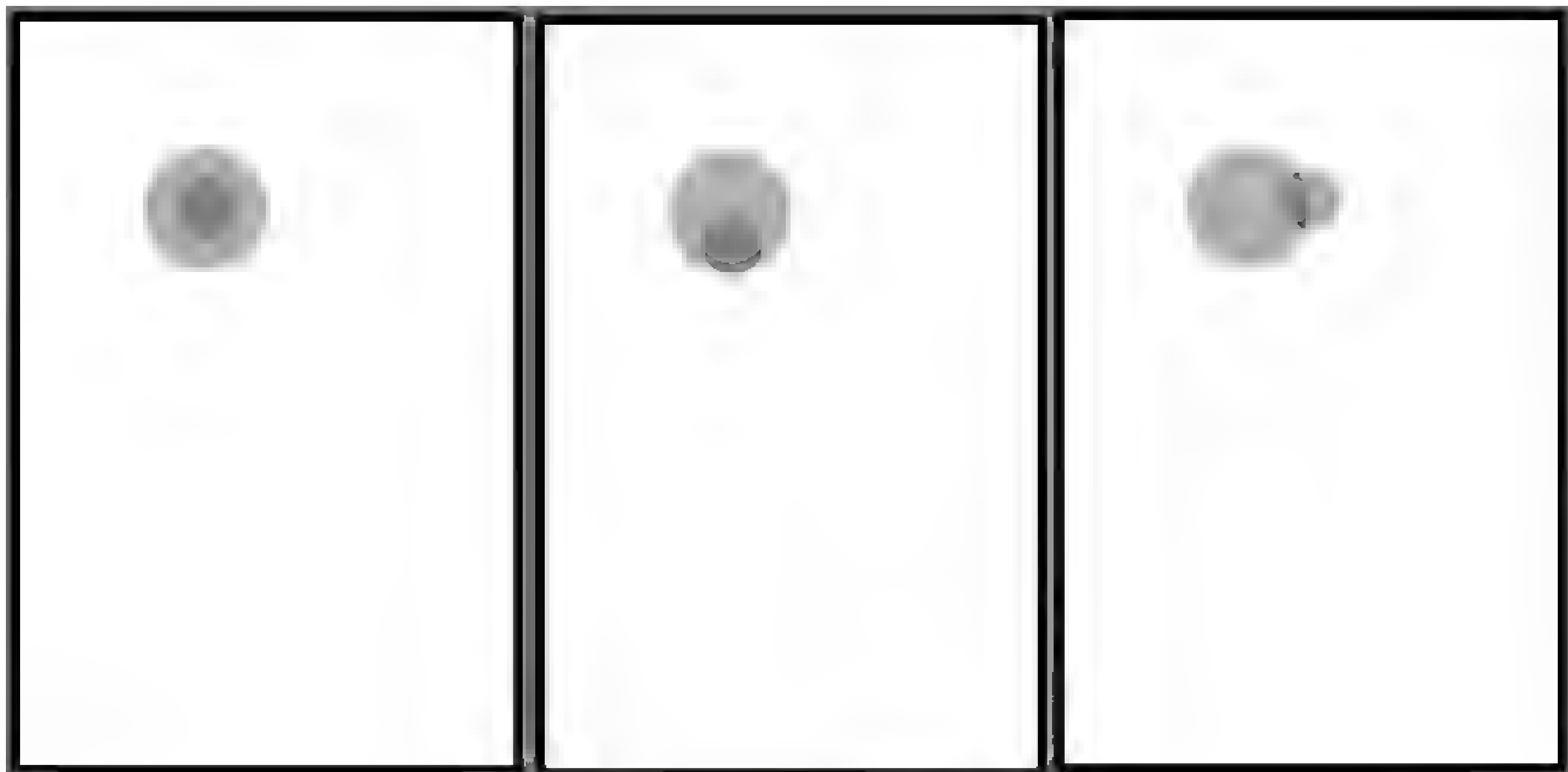


图 6-3　360° 平滑游戏摇杆

6.2 多触点实现图片缩放

在 Android SDK 2.0 中，对应 API 5 时开始支持屏幕的多触点，不过真机测试发现，目前最多支持两个触点；本小节就利用多触点的功能来实现缩放位图。

新建项目 “MoreContactsProject”，游戏框架为 SurfaceView 游戏框架，对应的项目源代码码为 “6-2（多触点缩放位图）”；这里需要注意，多触点是 API 5 以后支持的功能，所以 Android 模拟器要选择 SDK 5 或以上的版本，否则运行项目会报错。

首先定义用到的成员变量：

```
//声明一张 icon 位图
private Bitmap bmpIcon=BitmapFactory.decodeResource(this.getResources(),
                        R.drawable.icon);
//记录两个触屏点的坐标
private int x1, x2, y1, y2;
//倍率
private float rate = 1;
//记录上次的倍率
private float oldRate = 1;
//记录第一次触屏时线段的长度
private float oldLineDistance;
//判定是否头次多指触点屏幕
private boolean isFirst = true;
```

绘图函数：

```
public void myDraw() {
        ...
        canvas.save();
        //缩放画布(以图片中心点进行缩放，X、Y 轴缩放比例相同)
        canvas.scale(rate, rate, screenW / 2, screenH / 2);
        //绘制位图 icon
        canvas.drawBitmap(bmpIcon, screenW / 2 - bmpIcon.getWidth() / 2,
                                screenH / 2 - bmpIcon.getHeight() / 2, paint);
        canvas.restore();
        //便于观察，这里绘制两个触点时形成的线段
        canvas.drawLine(x1, y1, x2, y2, paint);
        ...
}
```

触屏监听事件：

```
public boolean onTouchEvent(MotionEvent event) {
        //用户手指抬起默认还原为第一次触屏标识位，并且保存本次的缩放比例
        if (event.getAction() == MotionEvent.ACTION_UP) {
                isFirst = true;
                oldRate = rate;
        } else {
                x1 = (int) event.getX(0);
                y1 = (int) event.getY(0);
                x2 = (int) event.getX(1);
                y2 = (int) event.getY(1);
                if (event.getPointerCount() == 2) {
                        if (isFirst) {
                                //得到第一次触屏时线段的长度
                                        oldLineDistance = (float)Math.sqrt
                                    (Math.pow(event.getX(1) - event.getX(0), 2) +
                                     Math.pow(event.getY(1) - event.getY(0), 2));
                                        isFirst = false;
                        } else {
                                //得到非第一次触屏时线段的长度
                                float newLineDistance = (float) Math.sqrt
                                    (Math.pow(event.getX(1) - event.getX(0), 2) +
                                     Math.pow(event.getY(1) - event.getY(0), 2));
                                //获取本次的缩放比例
                                rate = oldRate * newLineDistance/oldLineDistance;
                        }
                }
        }
        return true;
}
```

上面代码中，使用了两个常用的函数获取触屏点的 X、Y 坐标：

- MotionEvent.getX（int pointerIndex）
  作用：获取触屏点的 X 坐标
  参数：触屏点下标（下标从 0 开始）
- MotionEvent.getY（int pointerIndex）
  作用：获取触屏点的 Y 坐标
  参数：触屏点下标（下标从 0 开始）

除此之外，常用的方法还有获取触屏点的压力值函数：

- float getPressure（int pointerIndex）
  作用：获取触屏点的压力值
  参数：触屏点下标（下标从 0 开始）

运行项目，效果如图 6-4 所示。

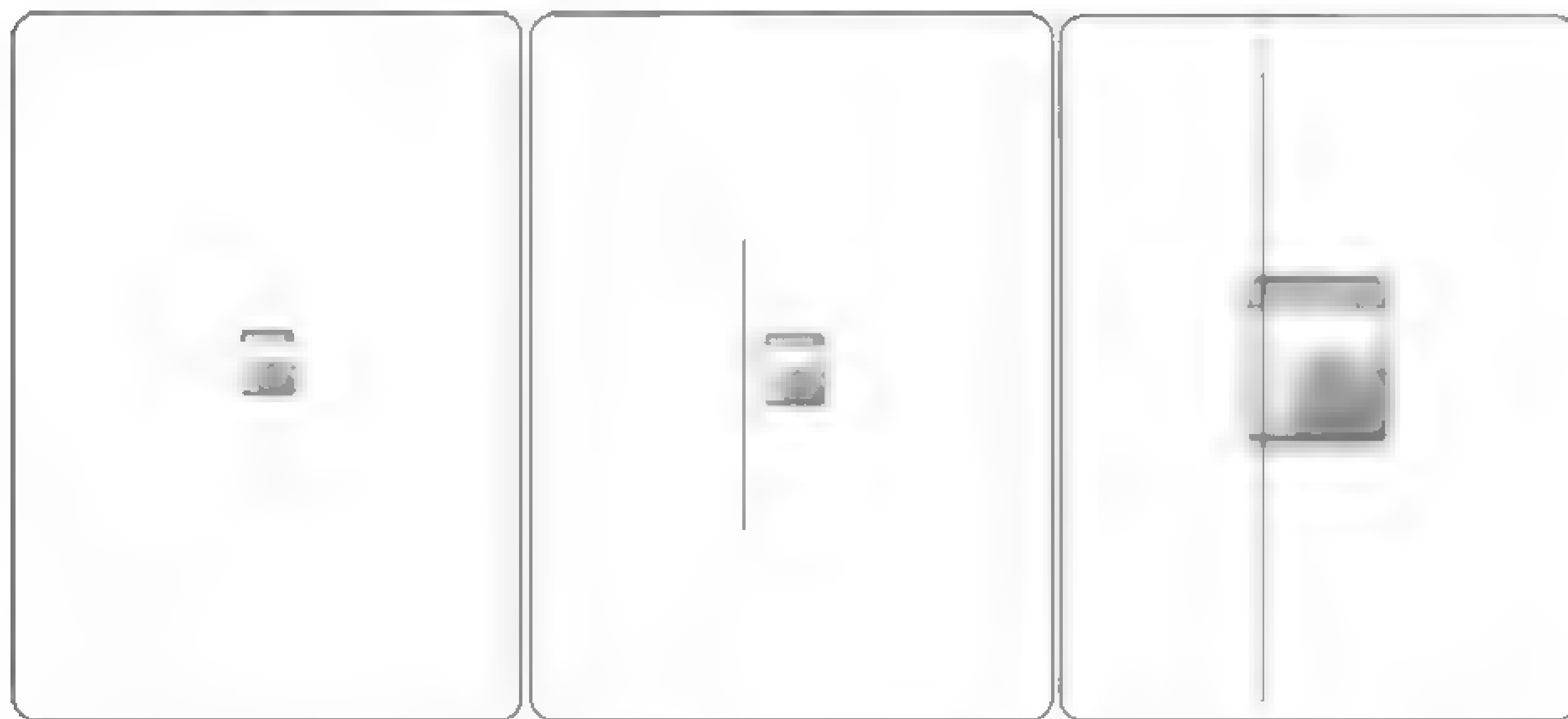


图 6-4　多点缩放位图

图 6-4 是由真机截图得到的效果，因为模拟器无法模拟多点触屏的效果。

6.3 触屏手势识别

所谓手势操作，类似跳舞机、Ezdancer 这些利用不同动作和音符让人手舞足蹈，Android 提供的的手势也让其平台的游戏和软件有了更多的花样操作和玩法。

手势的识别是根据玩家接触屏幕时间的长短、在屏幕上滑动的距离、按下抬起的时间等进行包装，其实就是 Android 对触屏事件监听做了包装和处理。

Android 的手势功能不光在软件开发中会用到，还经常应用在浏览器中的翻页，滚动页面等操作中；如果在开发 Android 游戏时，加上 Android 手势功能会让游戏增加一个亮点，比如一般的 CAG、PUZ 等类型的游戏选择关卡，简单背景的移动等都可以使用手势来操作。

如果需要监听当前用户的手势，首先需要一个监听接口：

```
GestureDetector.OnGestureListener
```

使用此接口，需要重写6个抽象函数，所有重写函数如下：

```
//按下
@Override
public boolean onDown(MotionEvent e) {
      return false;
}
// 短暂按下抬起
@Override
public boolean onSingleTapUp(MotionEvent e) {
      return false;
}
// 先短暂按下 、然后滑动、最后抬起
@Override
public boolean onFling(MotionEvent e1, MotionEvent e2, float
velocityX, float velocityY) {
      return false;
}
// 先短暂按下 、然后滑动
@Override
public boolean onScroll(MotionEvent e1, MotionEvent e2, float
distanceX, float distanceY) {
      return false;
}
// 先短暂按下 、短按不滑动
@Override
public void onShowPress(MotionEvent e) {
}
//长按不滑动
@Override
public void onLongPress(MotionEvent e) {
}
```

重写监听器的 6 个抽象函数如上都做了简单的解释，下面解释其重写函数中参数所表达的含义：

- e：保存触屏动作信息和触屏点坐标等；
- e1：保存触屏按下的动作信息和点坐标等；
- e2：保存触屏抬起的动作信息和点坐标等；
- velocityX：X 轴上的移动速度，像素/s；
- velocityY：Y 轴上的移动速度，像素/s；
- distanceX：现对于上次此事件触发，X 轴发生偏移量；

- distanceY：现对于上次此事件触发，Y 轴发生偏移量；
- distanceX 与 distanceY：千万不要理解为 e2 点 X 或 Y 坐标与 e1 点 X 或 Y 坐标的距离。

除了要使用手势监听器接口外，还需要一个触屏监听器接口：

```
View.OnTouchListener
```

使用触屏监听器接口还需重写一个抽象函数：

```
public boolean onTouch(View v, MotionEvent event) {
      return false;
}
```

一开始介绍手势时说过，所谓手势其实就是 Android 对触屏事件的封装，所以在监听用户的动作匹配哪种手势时，必须知道用户的触屏信息；而这个触屏信息，则由触屏监听器提供，用户的信息都存放在触屏监听器接口的抽象方法 onTouch 的参数中，所以应该修改 onTouch 函数如下：

```
public boolean onTouch(View v, MotionEvent event) {
      return GestureDetector.onTouchEvent(event);
}
```

实现了两个监听器之后，还要对当前的视图进行绑定。

手势类 GestureDetector 构造函数：

C GestureDetector.GestureDetector（OnGestureListener listener）

作用：为当前视图设置手势监听器

参数：手势监听器

为本视图设置触屏监听器：

M view.View.setOnTouchListener（OnTouchListener l）

作用：为当前 View 设置触屏监听器

参数：触屏监听器实例

下面简单地实现一个利用手势操作，匹配用户动作是：“先短暂按下，然后滑动，最后抬起（onFling）”，不断地在画布上绘制图片 icon 的小例子。

新建项目“GestureProject”，游戏框架为 SurfaceView 游戏框架，项目对应的源代码为“6-3（触屏手势识别）”。本视图使用触屏监听接口与手势监听接口，以及使用接口要重写的函数，如不需要修改的就不再贴出代码。

修改 MySurfaceView 类：

```
//检测手势的类
private GestureDetector gesture;
//保存所有添加的 icon 位图
private Vector<Bitmap> vec = new Vector<Bitmap>();
```

构造函数：

```
public MySurfaceView(Context context) {
      super(context);
      ...
      //实例 GestureDetector
      gesture = new GestureDetector(this);
      //为当前视图设置触屏监听器
      this.setOnTouchListener(this);
      ...
}
```

绘图函数：

```
public void myDraw() {
      ...
      for (int i = 0; i < vec.size(); i++) {
            canvas.drawBitmap(vec.elementAt(i), i * 5, 50, paint);
      }
      ...
}
```

手势 onFling 函数：

```
// 先短暂按下，然后滑动，最后抬起
public boolean onFling(MotionEvent e1, MotionEvent e2, float
velocityX, float velocityY) {
      //往图片容器里添加一个新的 icon 位图
      vec.add(BitmapFactory.decodeResource(this.getResources(),
R.drawable.icon));
      return false;
}
```

项目运行效果如图 6-5 所示。



图 6-5　触屏手势识别

6.4 加速度传感器

“传感器”一词对于接触过 Android 系统的人来说一定不会陌生，比如常见的赛车游戏，很多都使用了传感器的功能，不需要操作任何按键，只需要通过摇晃手机即可操控赛车的方向，这就是 Android 传感器中的一种加速度传感器。

加速度传感器又称为重力传感器，是 Android 系统提供传感器中的一种，除了加速度传感器外还有陀螺仪传感器、光传感器、恒定磁场传感器、方向传感器、恒定的压力传感器、接近传感器和温度传感器。

本章节通过详细讲解加速度传感器，让读者熟习实现一个传感器的过程与步骤，其他传感器的实现也都类似。

首先熟习几个有关传感器的类与接口。

1．SensorManager

传感器管理类，传感器的实例是通过传感器管理类来获取的；实例方式是通过系统提供的传感器服务获取传感器管理类实例的。

```
SensorManager  sm = (SensorManager) MainActivity;
instance.getSystemService (Service.SENSOR_SERVICE) ;
```

2．Sensor

传感器类，所有类型的传感器都包含在此类中；实例方式是通过传感器管理类来得的。

```
Sensor sensor = SensorManager.getDefaultSensor (int type) ;
```

代码中的参数 int type 指传感器的类型，其值是在 Sensor 类定义的静态常量。

- TYPE_ACCELEROMETER：加速度传感器（重力传感器）类型；
- TYPE_ALL：描述所有类型的传感器；
- TYPE_GYROSCOPE：陀螺仪传感器类型；
- TYPE_LIGHT：光传感器类型；
- TYPE_MAGNETIC_FIELD：恒定磁场传感器类型；
- TYPE_ORIENTATION：方向传感器类型；
- TYPE_PRESSURE：恒定压力传感器类型；
- TYPE_PROXIMITY：常量描述型接近传感器；
- TYPE_TEMPERATURE：温度传感器类型描述。

3．SensorEventListener

传感器监听接口，通过使用此接口可以监听当前传感器的属性及状态。

使用传感器监听器接口，需要实现以下两个函数：

```
//传感器获取值发生改变时响应此函数
public void onSensorChanged(SensorEvent event) {}
//传感器的精度发生改变时响应此函数
public void onAccuracyChanged(Sensor sensor, int accuracy) {}
```

一般传感器拥有 3 个值，除了 X、Y 外还有一个 Z 值，Z 值表示屏幕的朝向。这些值存放在 SensorEvent 的属性 value 数组中，获取的方法如下：

```
int x =  SensorEvent.values[0]; //手机横向翻滚
int y = SensorEvent.values[1]; //手机纵向翻滚
int z = SensorEvent.values[2]; //屏幕的朝向
```

- x>0 说明当前手机左翻，x<0 右翻；
- y>0 说明当前手机下翻，y<0 上翻；
- z>0 手机屏幕朝上，z<0 手机屏幕朝下。

注意：当前手机处于纵向还是横向，都会影响 X、Y 表示的意思!

如果当前手机是纵向屏幕：

- x>0 说明当前手机左翻，x<0 右翻；
- y>0 说明当前手机下翻，y<0 上翻。

如果当前手机是横向屏幕：

- x>0 说明当前手机下翻，x<0 上翻；
- y>0 说明当前手机右翻，y<0 左翻。

在使用了传感监听器后，还要为传感器注册上监听器才能完成监听：

SensorManager.registerListener(SensorEventListener listener,Sensor sensor, int rate)

作用：为传感器注册传感监听器；

第一个参数：传感监听器实例

第二个参数：需要监听的传感器实例

第三个参数：监听传感器速率类型

监听器的速率类型如下：

- SENSOR_DELAY_NORMAL：正常；
- SENSOR_DELAY_UI：适合界面；
- SENSOR_DELAY_GAME：适用于游戏；
- SENSOR_DELAY_FASTEST：最快。

下面利用加速度传感器实现一个操作圆形小球移动的小游戏。

新建项目“SensorProject”，游戏框架为 SurfaceView 游戏框架，对应的源代码为“6-4

（加速度传感器）”。

修改 MySurfaceView，首先定义一些用到的成员变量：

```
//声明一个传感器管理器
private SensorManager sm;
//声明一个传感器
private Sensor sensor;
//声明一个传感器监听器
private SensorEventListener mySensorListener;
//圆形的 x、y 坐标
private int arc_x, arc_y;
//传感器的 x、y、z 值
private float x = 0, y = 0, z = 0;
```

构造函数：

```
public MySurfaceView(Context context) {
      ...
      //获取传感器管理类实例
      sm = (SensorManager) MainActivity.instance.
      getSystemService(Service.SENSOR_SERVICE);
      //实例一个重力传感器实例
      sensor = sm.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
      //实例传感器监听器
      mySensorListener = new SensorEventListener() {
      @Override
      //传感器获取值发生改变时在响应此函数
      public void onSensorChanged(SensorEvent event) {
            x = event.values[0]; //手机横向翻滚
            //x>0 说明当前手机左翻，x<0 右翻
            y = event.values[1]; //手机纵向翻滚
            //y>0 说明当前手机下翻，y<0 上翻
            z = event.values[2]; //屏幕的朝向
            //z>0 手机屏幕朝上，z<0 手机屏幕朝下
            arc_x -= x;
            arc_y += y;
      }
      @Override
      //传感器的精度发生改变时响应此函数
      public void onAccuracyChanged(Sensor sensor, int accuracy) {
      }
      };
      //为传感器注册监听器
      sm.registerListener(mySensorListener, sensor, SensorManager.
                                         SENSOR_DELAY_GAME);
      ...
```

```
}
```

绘图函数:

```
public void myDraw() {
      ...
      paint.setColor(Color.RED);
      canvas.drawArc(new RectF(arc_x, arc_y, arc_x + 50, arc_y + 50),
                              0, 360, true, paint);
      paint.setColor(Color.YELLOW);
      canvas.drawText("当前重力传感器的值:", arc_x - 50, arc_y - 30, paint);
      canvas.drawText("x=" + x + ",y=" + y + ",z=" + z, arc_x - 50, arc_y,
paint);
      String temp_str = "Himi 提示:  ";
      String temp_str2 = "";
      String temp_str3 = "";
      if (x < 1 && x > -1 && y < 1 && y > -1) {
            temp_str += "当前手机处于水平放置的状态";
            if (z > 0) {
                  temp_str2 += "并且屏幕朝上";
            } else {
                  temp_str2 += "并且屏幕朝下,提示别躺着玩手机, 对眼睛不好哟";
            }
      } else {
            if (x > 1) {
                  temp_str2 += "当前手机处于向左翻的状态";
            } else if (x < -1) {
                  temp_str2 += "当前手机处于向右翻的状态";
            }
            if (y > 1) {
                  temp_str2 += "当前手机处于向下翻的状态";
            } else if (y < -1) {
                  temp_str2 += "当前手机处于向上翻的状态";
            }
            if (z > 0) {
                  temp_str3 += "并且屏幕朝上";
            } else {
                  temp_str3 += "并且屏幕朝下,提示别躺着玩手机, 对眼睛不好哟";
            }
      }
      paint.setTextSize(10);
      canvas.drawText(temp_str, 0, 50, paint);
      canvas.drawText(temp_str2, 0, 80, paint);
      canvas.drawText(temp_str3, 0, 110, paint);
      ...
}
```

项目运行效果如图 6-6 所示。

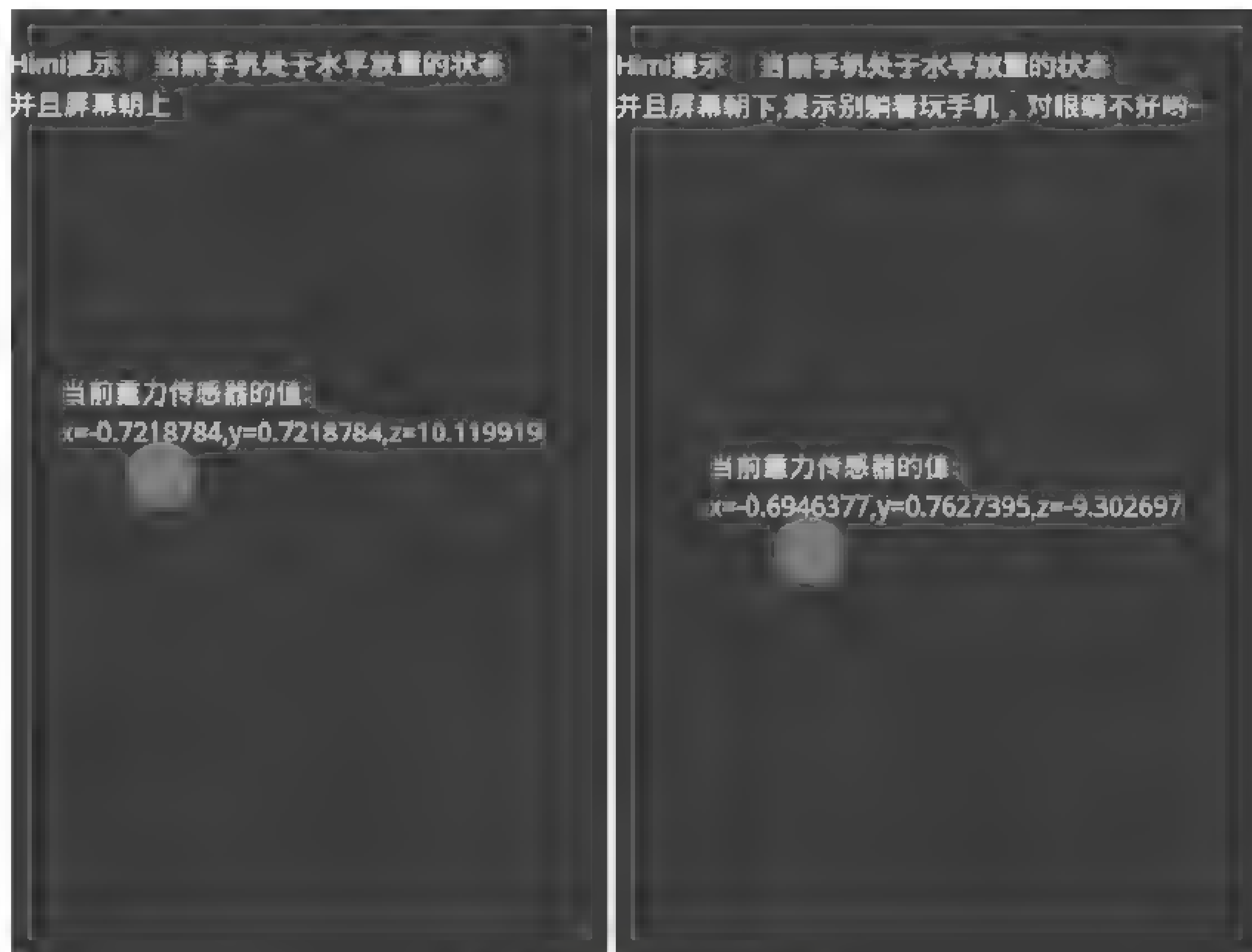


图 6-6　加速度传感器

6.5　9patch 工具的使用

9patch: 是一个对 png 图片做处理的工具，能够生成“*.9.png”格式的图片；

“*.9.png”格式是 Android OS 支持的一种特殊的图片格式，用它可以实现部分拉伸。因为是经过“9patch 工具”进行特殊处理过的，所以能很好地解决一般 png 格式图拉伸会失真、拉伸不正常的问题。

在 Android SDK 路径下的 tools 目录下的“draw9patch.bat”，就是 9patch 工具，官方名为 NinePatch。双击“draw9patch.bat”则启动 9patch 工具，界面如图 6-7 所示。

单击菜单“file”选项，选中“Open 9-patch…”选项，导入一张 png 图片，然后真正进入 9patch 的操作界面（如图 6-8 所示）：

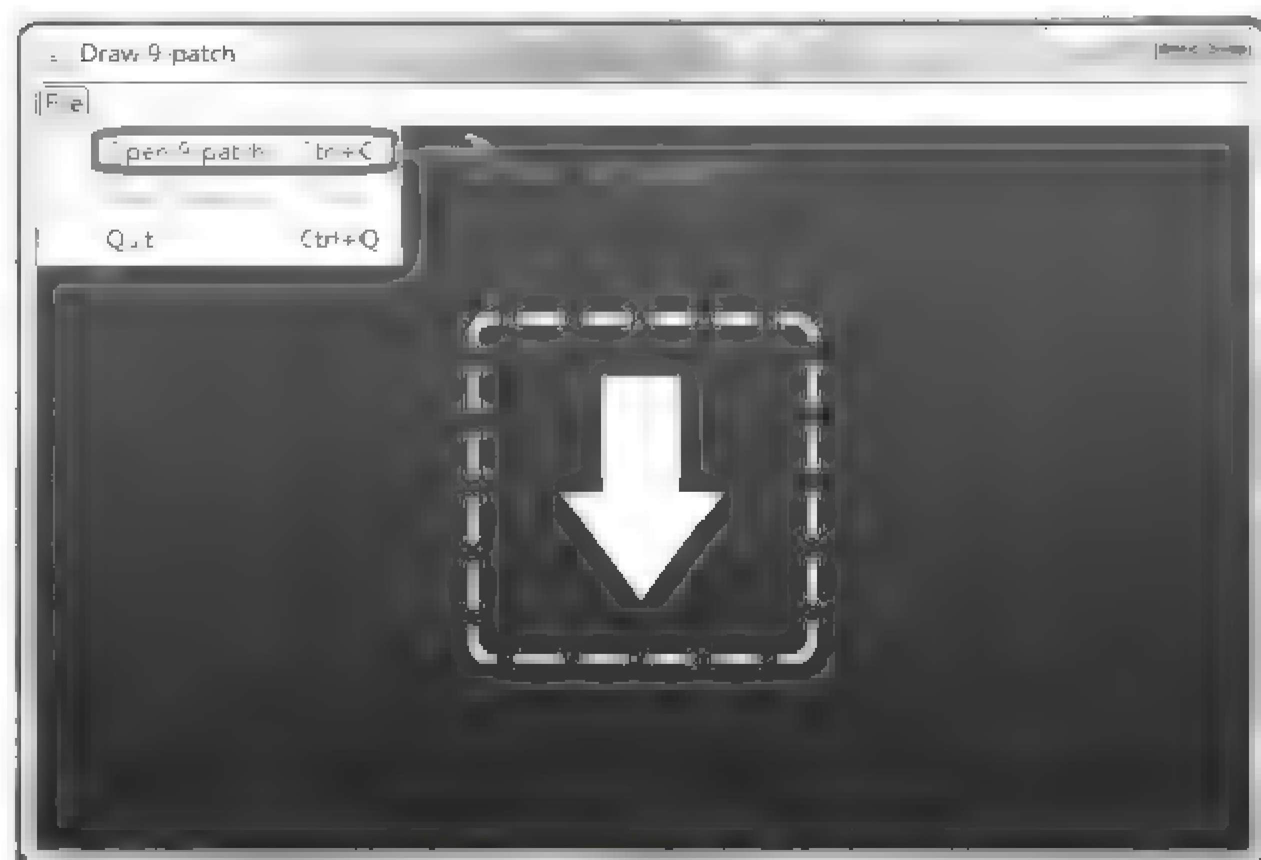


图 6-7　9patch 工具主界面

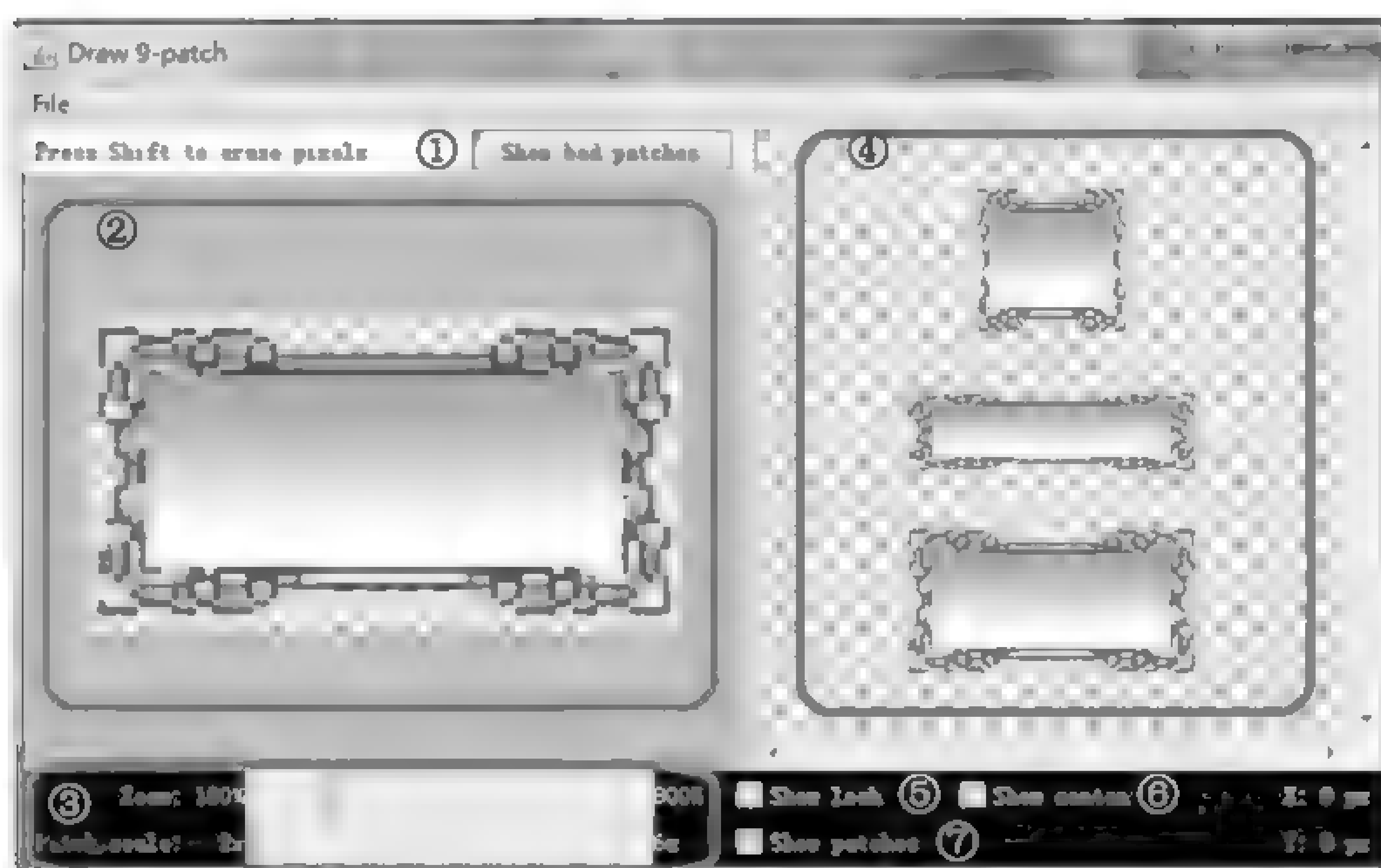


图 6-8　9patch 操作界面

序号①：在拉伸区域周围用红色边框显示拉伸后的图片可能会产生变形的区域，如果完全消除该内容则图片拉伸后是没有变形的，也就是说，不管如何缩放图片显示都是良好的（实际测试发现 NinePatch 编辑器是根据图片的颜色值来区分是否为 bad patch 的，一般只要色差不是太大不用考虑这个设置）。

序号②：是导入的图片，以及可操作区域。

序号③：这里 zoom 的长条 bar 是对导入的图进行放大缩小操作，这里的放大缩小只是为了让使用者更方便，毕竟对像素点操作比较费神，下面的 patch scale 是序列 ④区域中的 3 种形态拉伸后的一个预览操作，可以看到图片拉伸后的效果。

序号④：从上到下依次为纵向拉伸的效果预览、横向拉伸的效果预览以及整体拉伸的效果预览。

序号⑤：如果勾选上，那么当鼠标放在 ② 区域内的时候，当前位置为不可操作区域且会出现一张 lock 的图。

序号⑥：如果勾选上，那么在④区域中就会看到当前操作的像素点在拉伸预览图中的相对位置和效果。

序号⑦：在编辑区域显示图片拉伸的区域。

操作：鼠标左键选取需要拉伸的像素点，shift+鼠标左键取消当前像素点。

操作区域：如图 6-9 所示。

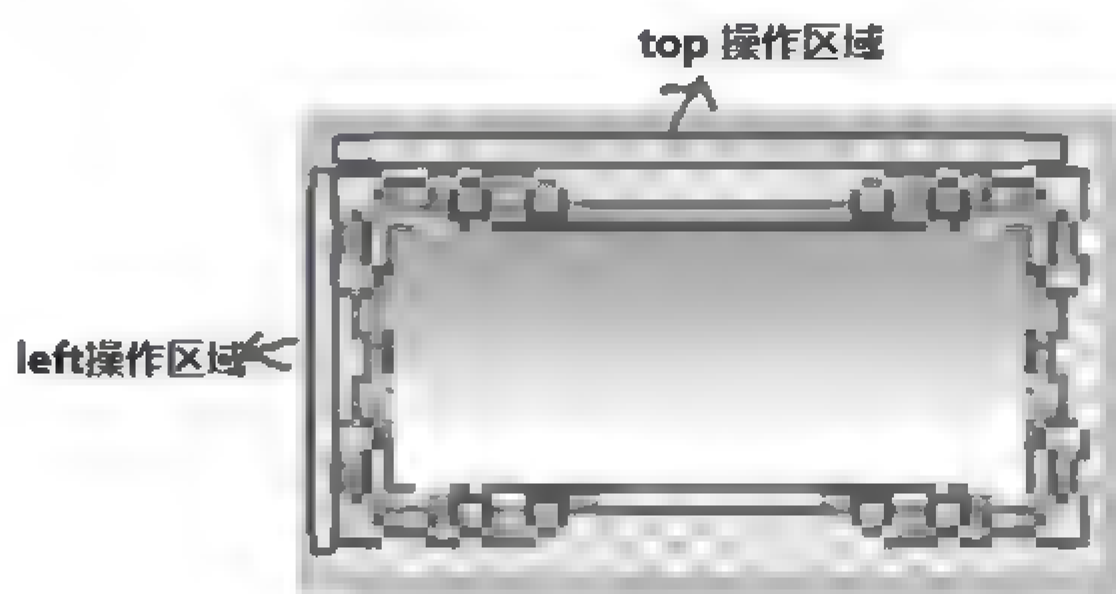


图 6-9　编辑操作区域

从图 6-9 看到导入的 png 图片默认周围多了一圈像素点，这一圈像素点就是可操作区域。因为下方和右方可操作区域是指定内容的显示区域，属于可选区域，可不予理会，但是要注意内容区域的标记不能有间断，也就是说标记要连续且仅有一处，否则“.9.png”图片在放入项目下会报错。

注意 left 和 top 操作区域。top 操作区域的一排像素点，表示横向拉伸的像素点；left 操作区域的一排像素点，表示纵向拉伸的像素点。

代码中加载一张“.9.png”图的步骤如下：

步骤1 先把“.9.png”图资源生成一张 Bitmap 位图，与正常图片加载方式一样；

步骤2 绘制时候利用 NinePatch 类进行绘制

使用“*.9.png”的好处

1. 省精力和时间

如果有一张 50×50 的类似上面那种带花边的 png 图片，那么在 Android 或者大分辨率的机器上使用的话，肯定需要让美工给重新做一张，而通过 9patch 处理得到的“*.9.png”就会省去美工不少工作。

2. 省内存

如果不想用代码对小图进行缩放以再次使用（因为考虑会失真），可能会多加图片，这样一来游戏包的大小就会增加了，几 KB 到几十 KB 不等，而利用 9patch 处理就省去了这些麻烦，节省了空间。

3. 减少代码量

有些人会说，我用代码一样能实现图片的效果不失真，没错，利用设置可视区域等代码可以实现，但可以肯定的是没有用“.9.png”的方式来的这么简单方便！

新建项目“NinePatchProject”，游戏框架为 SurfaceView 游戏框架，对应的源代码为“6-5（9patch 工具）”。

首先导入两张图片资源（如图 6-10 所示）：

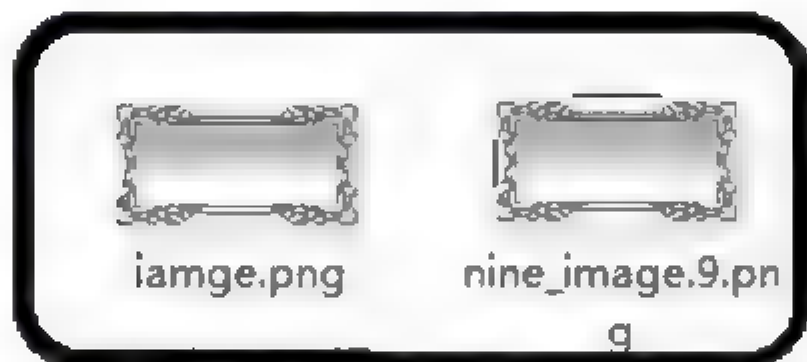


图 6-10　图片资源

图 6-10 中左侧是一张正常的 png 图片，右侧是通过 9patch 工具处理过的“.9.png”图。

声明用到的成员变量：

```
//原图
private Bitmap bmp_old;
//通过 9patch 工具生成的".9.png"图
private Bitmap bmp_9path;
//声明 NinePatch
private NinePatch np;
```

构造函数：

```
public MySurfaceView(Context context) {
      …
      //将图片资源生成位图
      bmp_old = BitmapFactory.decodeResource(getResources(),
                                    R.drawable.iamge);
      bmp_9path = BitmapFactory.decodeResource(getResources(),
                                    R.drawable.nine_image);
      //实例 NinePatch 实例
      np = new NinePatch(bmp_9path, bmp_9path.getNinePatchChunk(), null);
      …
}
```

NinePatch 构造函数：

NinePatch (Bitmap bitmap, byte[] chunk, String srcName)

第一个参数：“.9.png”的位图实例

第二个参数：参数其实要求传入处理拉伸方式，当然不需要自己传入，因为“.9.png”图片自身有这些信息数据，也就是用 9patch 工具操作的信息！这个参数直

接用“.9.png”图片自身的数据调用 getNinePatchChunk()采用自身操作信息即可。

第三个参数：图片源的名称，这个参数为可选参数，直接 null 就可以。

绘图函数：

```
public void myDraw() {
    …
    canvas.drawColor(Color.BLACK);
    //---------这里是为了更好地展示出缩放拉伸后的区别
    RectF rectf_old_two = new RectF(0, 50, bmp_old.getWidth() *
                            2, 120 + bmp_old.getHeight() * 2);
    RectF rectf_old_third = new RectF(0, 120 + bmp_old.getHeight()
            * 2, bmp_old.getWidth() * 3, 140 + bmp_old.getHeight() * 2 +
            bmp_old.getHeight()* 3);
    // --------下面是对正常 png 绘画方法-----------
    canvas.drawBitmap(bmp_old, 0, 0, paint);
    canvas.drawBitmap(bmp_old, null, rectf_old_two, paint);
    canvas.drawBitmap(bmp_old, null, rectf_old_third, paint);
    RectF rectf_9path_two = new RectF(250, 50, 250 +
            bmp_9path.getWidth() * 2, 90 + bmp_9path.getHeight() * 2);
    RectF rectf_9path_third = new RectF(250, 120 +
            bmp_9path.getHeight() * 2, 250 + bmp_9path.getWidth() * 3,
            140 + bmp_9path.getHeight() * 2+ bmp_9path.getHeight()*3);
    canvas.drawBitmap(bmp_9path, 250, 0, paint);
    // --------下面是".9.png"图像的绘画方法-----------
    np.draw(canvas, rectf_9path_two);
    np.draw(canvas, rectf_9path_third);
    …
}
```

代码中使用了绘制位图的方法：

M NinePatch.draw（Canvas canvas, RectF location)

作用：绘制“.9.png”位图

第一个参数：画布实例

第二个参数：RectF 实例

项目效果如图 6-11 所示。

最后需要提醒一点：千万不要随便将一个“.png”格式的图片改名成“.9.png”格式，因为不是通过 9patch 生成的“.9.png”格式的图导入在项目中时，项目会报错，ADT 会检测资源文件“.9.png”图是否由 9patch 生成。

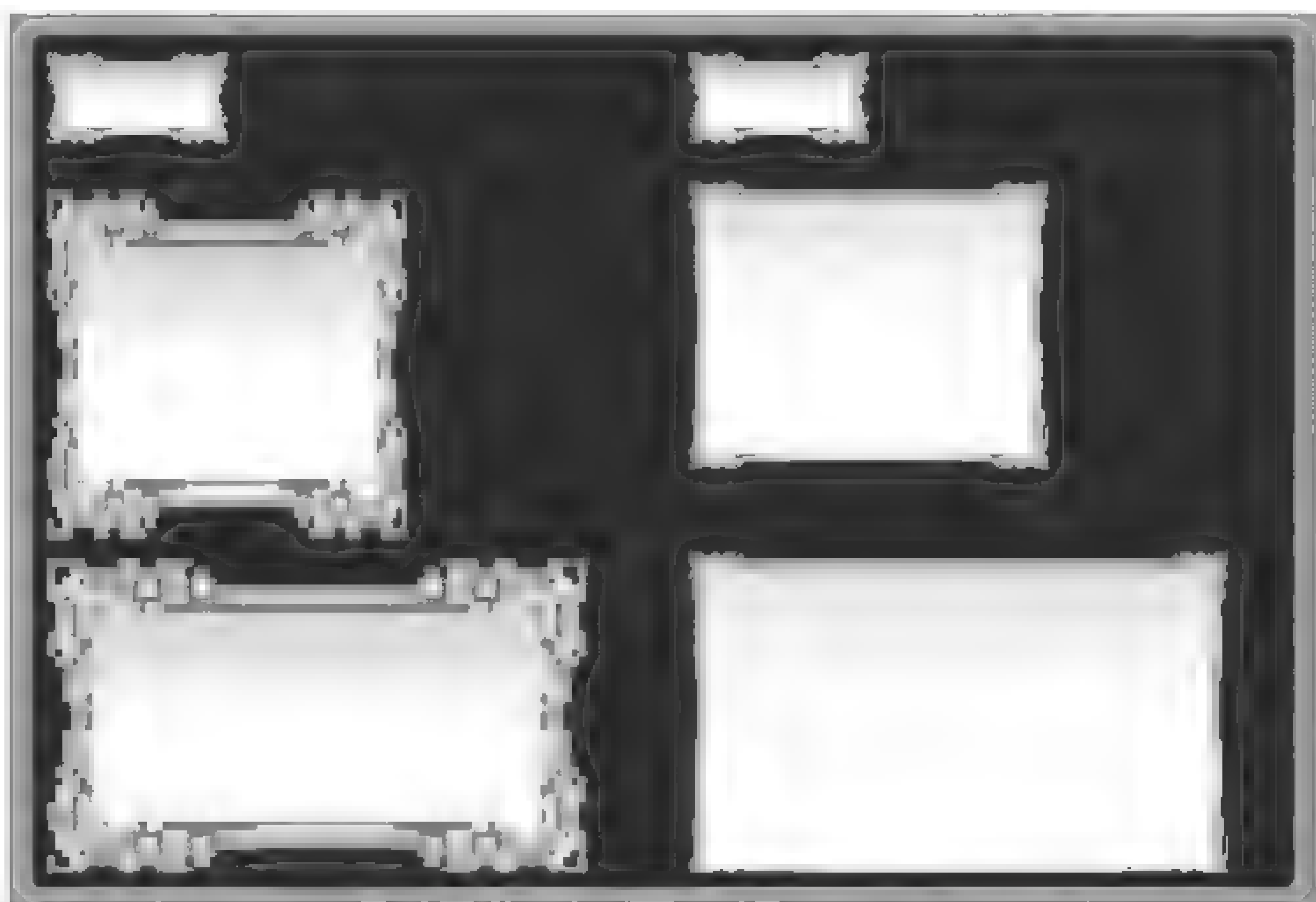


图 6-11 png 与.9.png 拉伸效果对比图

6.6 代码实现截屏功能

在体育类的游戏中，例如篮球游戏，有时需要实现一个摄像头功能，模拟现场直播的效果。除此之外在手机网游中很多时候也需要截屏功能，用户可以将游戏界面截图下来，然后将游戏的截图分享或者上传等。

这一小节主要介绍在游戏中常用的代码实现截屏的方法。通过之前的游戏基础的章节学习可以知道，想在视图中显示绘制的图形或者位图等，都需要得到一个画布，然后通过画布再进行绘制。截屏的实现原理，其实就是通过手动创建一张位图，然后通过此位图得到一个Canvas（画布）实例，接着利用得到的这个画布进行绘制，而画布上绘制的这些图形其实都保存在最初创建的位图上。最后只要利用游戏主画布绘制这张位图即可。

新建项目“ClipScreenProject”，游戏框架为 SurfaceView 框架，对应的源代码为“6-6（截屏）”。

修改 MySurfaceView 类：

```
//icon 位图
private Bitmap bmpIcon;
//截屏的图
private Bitmap bmpClip;
//截屏的画布
private Canvas canvasClip;
```

视图创建函数：

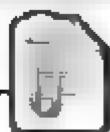
```
public void surfaceCreated(SurfaceHolder holder) {
      ...
      //实例 Icon 位图
      bmpIcon = BitmapFactory.decodeResource
                    (this.getResources(), R.drawable.icon);
      //创建一个与当前屏幕大小相同的图片
      bmpClip = Bitmap.createBitmap(this.getWidth(), this.
                    getHeight(), Config.ARGB_8888);
      //通过创建的图片，得到画布实例
      canvasClip = new Canvas(bmpClip);
      //利用截屏画布刷屏
      canvasClip.drawColor(Color.WHITE);
      //利用画布绘制一张 icon 图
      canvasClip.drawBitmap(bmpIcon, 0, 0, paint);
      ...
}
```

由于 this.getWidth()、this.getHeight()得到视图宽高的函数，必须在视图创建后才能正常得到其值，所以这里创建截屏位图的代码写在此函数中。

- Bitmap.createBitmap（int width, int height, Config config）
  作用：创建一张位图
  第一个参数：位图宽
  第二个参数：位图高
  第三个参数：图片配置信息
- Canvas.Canvas（Bitmap bitmap）
  作用：得到位图的画布实例
  第一个参数：位图实例

绘图函数：

```
public void myDraw() {
      ...
      //绘制 icon 位图
      canvas.drawBitmap(bmpIcon, 0, 0, paint);
      //绘制截屏的位图
      canvas.drawBitmap(bmpClip, 50, 50, paint);
      ...
}
```



注意

绘制截屏的位图，其实就是绘制截屏的画布，而截屏中绘制的 icon 位图相对于截屏画布的坐标仍然是（0，0）。

项目截图效果如图 6-12 所示。



图 6-12　截屏效果图

图 6-12 白色部分与视图大小相同，但是从屏幕无法完整地看出整个截屏画布的大小，为便于观察，截屏画布并没有与主画布重合。

6.7 效率检视工具

程序都是改出来的，而流畅的程序则是优化出来的，手机的应用程序也是如此。不管是网游还是单机游戏还是应用，对于程序的内存和代码优化都是必须重视的。

那么优化代码应该如何下手？程序中哪段代码或者哪个函数占用了更多的运行时间？Android SDK 为开发者提供了一个效率检视工具——TraceView！

TraceView 是 Android 平台提供给开发者的一个很好的性能分析工具；通过 TraceView 提供的图形化方式不仅让开发者更好地跟踪程序的性能，而且还能具体到每个函数的性能信息。

使用 TraceView 检测一个程序的效率，首先需要得到此程序的追踪文件。得到程序的追踪文件很简单，只要调用两个函数即可。

1．开始记录函数

M Debug.*startMethodTracing*（String traceName）;

作用：开始记录程序运行信息，并且在“/sdcard”目录下生成一个追踪文件

“traceName.trace”

参数：追踪文件的文件名

M Debug.*startMethodTracing*();

作用：开始记录程序运行信息，并且默认在“/sdcard”目录下生成一个追踪文件“dmtrace.trace”

2. 停止记录函数

M Debug.stopMethodTracing();

作用：停止记录程序运行信息

当开始记录函数被调用之后，就会生成此追踪文件，但是在停止记录函数被调用之前，追踪文件大小一直为 0，没有任何数据，只有停止记录函数被调用之后，系统才会将记录的数据存放在监测文件中。

生成追踪文件需要注意以下几点：

（1）必须保证 Android 模拟器或者真机设备中有 SDCard，因为生成的追踪文件默认保存在 SDCard 中。

（2）因为追踪文件默认保存在 SDCard 中，且当停止记录后会把记录的数据存放在此文件中，所以在 Mainfestmani.xml 中添加写入权限：

```
<uses-permission
android:name="android.permission.WRITE_EXTERNAL_STORAGE">
</uses-permission>
```

（3）如果 SDCard 空间太小，则程序的追踪文件记录到 SDCard 容量满时就停止记录，所以为了生成一个正确的追踪文件，生成一个存储空间适合的 SDCard 也较为重要。毕竟记录的时间越长，追踪文件也就越大。

当得到程序的追踪文件后，将追踪文件导出到电脑中。这里编者做了一个简单的测试程序，生成了一个名为“himi.trace”的追踪文件，并且从手机 SDCard 中导出存放在 C 盘根目录下，那么通过 DOS 命令启动 TraceView 来解析这个追踪文件：

```
C:\Users\Himi>traceview c:\himi
```

有时输入命令后，并没有打开 TraceView 窗口，而是报出如图 6-13 所示的错误。

```
C:\Users\Himi>traceview c:\himi
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
        at java.util.Arrays.copyOf(Unknown Source)
        at java.util.Arrays.copyOf(Unknown Source)
        at java.util.ArrayList.ensureCapacity(Unknown Source)
        at java.util.ArrayList.add(Unknown Source)
        at com.android.traceview.DmTraceReader.getThreadTimeRecords(DmTraceReade
r.java:527)
        at com.android.traceview.TimeLineView.<init>(TimeLineView.java:316)
        at com.android.traceview.MainWindow.createContents(MainWindow.java:93)
        at org.eclipse.jface.window.Window.create(Window.java:431)
        at org.eclipse.jface.window.Window.open(Window.java:790)
        at com.android.traceview.MainWindow.run(MainWindow.java:58)
        at com.android.traceview.MainWindow.main(MainWindow.java:190)
```

图 6-13　DOS 命令异常截图

这个错误的解决方案是：到 SDK 路径的 tools 目录下找到 traceview.bat 文件，单击鼠标右键，选中“编辑”（或者记事本打开）选项，然后将其中的最后一行替换如下：

```
call java -Xms128m -Xmx512m -Djava.ext.dirs=%javaextdirs% -jar %jarpath%%*
```

以上的解决方案就是将 TraceView 的运行内存增大，避免出现内存溢出错误。

正常情况下，在 DOS 界面输入命令后，等待 2s 左右，TraceView 界面就会出现，如图 6-14 所示。

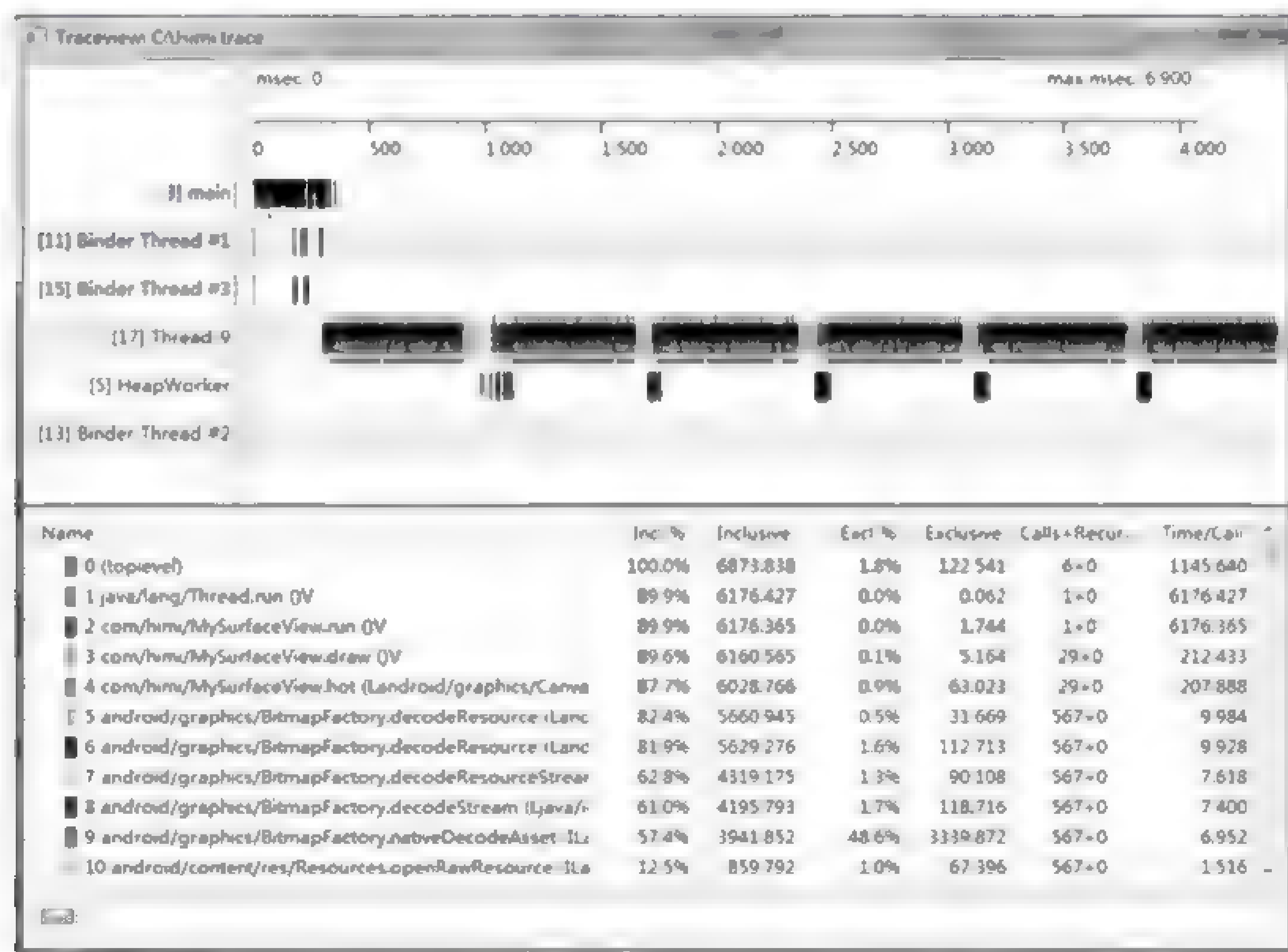


图 6-14　TraceView 效率检视工具

这里对一些单词含义进行简单的介绍：

- Exclusive: 同级函数本身运行的时间。

- Inclusive：除统计函数本身运行的时间外再加上调用子函数所运行的时间。
- Name：列出的是所有的调用项，前面的数字是编号，展开可以看到有 Parent 和 Children 子项，就是指被调用和调用。
- Incl%: inclusive 时间占总时间的百分比。
- Excl%: 执行占总时间的百分比。
- Time/Call: 总的时间（ms）。
- Calls+Recur Calls/Total: 调用和重复调用的次数。
- max msec：追踪文件记录的总时间。

从图 6-14 所示的 TraceView 界面中，可以看到有各种颜色，每种颜色代表不同的函数和步骤。同一颜色的区域越大，就代表这个步骤运行时间越长。

在图 6-14 下面的统计表中，明显可以看出除了序列 0、1 是系统函数外，序列 2 和 3 函数占用的时间比较长（因为是游戏的主逻辑和主绘图函数）。仔细观察序列 4，它是个自定义的函数，名为“hot”，它占用的运行时间几乎与主线程/主函数的时间一样了，那么肯定有必要到程序中查看一下此函数的实现方法。其实这个方法是特意编写的，就是为了来演示 TraceView，这个 hot 函数的代码如下：

```
public void hot(Canvas canvas) {
    for (int i = 1; i < 100; i++) {
        Bitmap bmp = BitmapFactory.decodeResource(getResources(),
                R.drawable.icon);
        canvas.drawBitmap(bmp, i += 2, i += 2, paint);
    }
}
```

这个 hot 函数在每次调用时，都会在画布上绘制 100 张 icon 图，所以占用了较多的运行时间。

通过上述讲解与简单的测试可以看出，TraceView 是个非常好的程序监视工具，它通过追踪文件生成的图形分析与函数列表信息，可以很直观地帮助开发人员找出程序运行缓慢的原因，让代码不断改进，程序不断完善。

6.8 游戏视图与系统组件共同显示

在游戏开发中，有时想直接使用 Android 提供的组件，比如按钮、文本编辑框等，游戏是在 view 视图中进行的，那么添加系统组件的话，就应该想到将游戏自定义的视图 view 作为一个组件，与系统组件一同放在布局中，然后通过 Activity 显示布局一并显示在手机屏幕上。

大概想到这些之后，下面就来简单实现一个手机屏幕既显示游戏 view，也显示系统组件的例子。

新建项目“ViewAndItem”，游戏框架为 SurfaceView 游戏框架，项目对应的源代码为“6-8（游戏视图与系统组件）”。既然游戏视图与系统组件都是由布局一起显示，那么首先修改 MainActivity 类：

```
public class MainActivity extends Activity {
      @Override
      public void onCreate(Bundle savedInstanceState) {
             super.onCreate(savedInstanceState);
             //设置全屏
             this.getWindow().setFlags(WindowManager.LayoutParams.
                        FLAG_FULLSCREEN, WindowManager.LayoutParams.
                        FLAG_FULLSCREEN);
             requestWindowFeature(Window.FEATURE_NO_TITLE);
             //显示 main.xml 布局文件
             setContentView(R.layout.main);
      }
}
```

然后修改 res/layout 下的 main.xml 布局文件如下：

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
android:orientation="vertical" android:layout_width="fill_parent"
      android:layout_height="fill_parent">
      <!-- 自定义的 SurfaceView 视图组件 -->
      <com.vai.MySurfaceView android:layout_width="wrap_content"
         android:layout_height="wrap_content" android:text="@string/hello"
         android:id="@+id/myview" />
      <Button android:layout_width="wrap_content"
          android:layout_height="wrap_content" android:text="Button"
           android:layout_alignParentBottom="true" />
</RelativeLayout>
```

整个布局使用相对布局，其中包含自定义的 SurfaceView 视图和系统 Button 按钮；最后一步，也是最重要的一步，修改 MySurfaceView 的构造函数如下：

```
public MySurfaceView(Context context, AttributeSet attrs) {
      super(context, attrs);
}
```

MySurfaceView 构造函数的第二个参数指自定义的组件的一些属性，如组件的长宽等，自定义组件的属性其实就是通过这个参数进行传递的。

项目运行效果如图 6-15 所示。



图 6-15　游戏视图与系统组件

6.9 蓝牙对战游戏

由于 Android 模拟器中没有蓝牙模块，请读者在真机设备上进行测试，且蓝牙模块只有在 Android SDK 2.0（API 5）以上（含）版本才开始支持。

首先介绍 BluetoothAdapter（蓝牙适配器类），这个类对蓝牙当前的状态是否可见、是否已打开等进行设置与监听。BluetoothAdapter 的实例方法如下：

```
BluetoothAdapter ba = BluetoothAdapter.getDefaultAdapter();
```

BluetoothAdapter 的常用函数如下：

1．int BluetoothAdapter.getState()

作用：获取当前终端设备蓝牙状态

返回值：int 常量值

蓝牙开关常量如下：

- BluetoothAdapter.STATE_OFF = 10 [0xa]
- BluetoothAdapter.STATE_ON = 12 [0xc]

2. BluetoothAdapter.enable()

作用：启动蓝牙

3. BluetoothAdapter.disable()

作用：关闭蓝牙

4. BluetoothAdapter.startDiscovery()

作用：启动蓝牙可被发现

5. BluetoothAdapter.getAddress()

作用：得到当前设备蓝牙地址

BluetoothDevice 是蓝牙设备类，其构造方法如下：

C BluetoothDevice bd = BluetoothAdapter.getRemoteDevice（String address）;

作用：获取一个设备实例

参数：设备地址

BluetoothSocket 是蓝牙连接类，用于发送与接受报文数据，其构造方法如下：

C BluetoothSocket bs=BluetoothDevice.createRfcommSocketToServiceRecord(UUID uuid)

作用：获取一个蓝牙连接实例

参数：UUID 实例

BluetoothDevice 的常用函数如下：

1. BluetoothSocket.connect()

作用：与此蓝牙设备创建连接

2. BluetoothSocket.getInputStream()

作用：得到一个输入流，用于（监听）获取报文数据

此方法对已配对的蓝牙设备进行监听报文数据，如接收到数据则会创建得到一个输入流实例，用于读取报文数据；如接收不到报文数据则默认阻塞。

3. BluetoothSocket.getOutputStream()

作用：得到一个输出流，用于发送报文数据

了解这些类之后，下面就通过项目实例详细讲解蓝牙功能的开发。

一般开发蓝牙需要两台真机设备，这里首先讲解如何利用 IVT 软件与蓝牙适配器模拟一个蓝牙终端（本节内容基于 Windows 7 系统）。

首先安装 Bluesoleil 软件，成功安装之后，将蓝牙适配器连接到电脑，然后右击任务栏的 IVT 图标，选中“显示经典界面”选项，然后在 IVT 主菜单中依次找到“蓝牙→我的属性设备→串口”观察当前启动的串口，记住默认启动的是 com3 与 com4 串口，以便后续的配置。

然后安装 Windows NT 软件，成功安装后单击 Windows NT 目录下的“hypertrm.exe”文件，出现位置信息界面，如图 6-16 所示。

输入地区区号，然后直接单击“确定”按钮，进入下一步“新建连接”窗口，如图 6-17 所示。

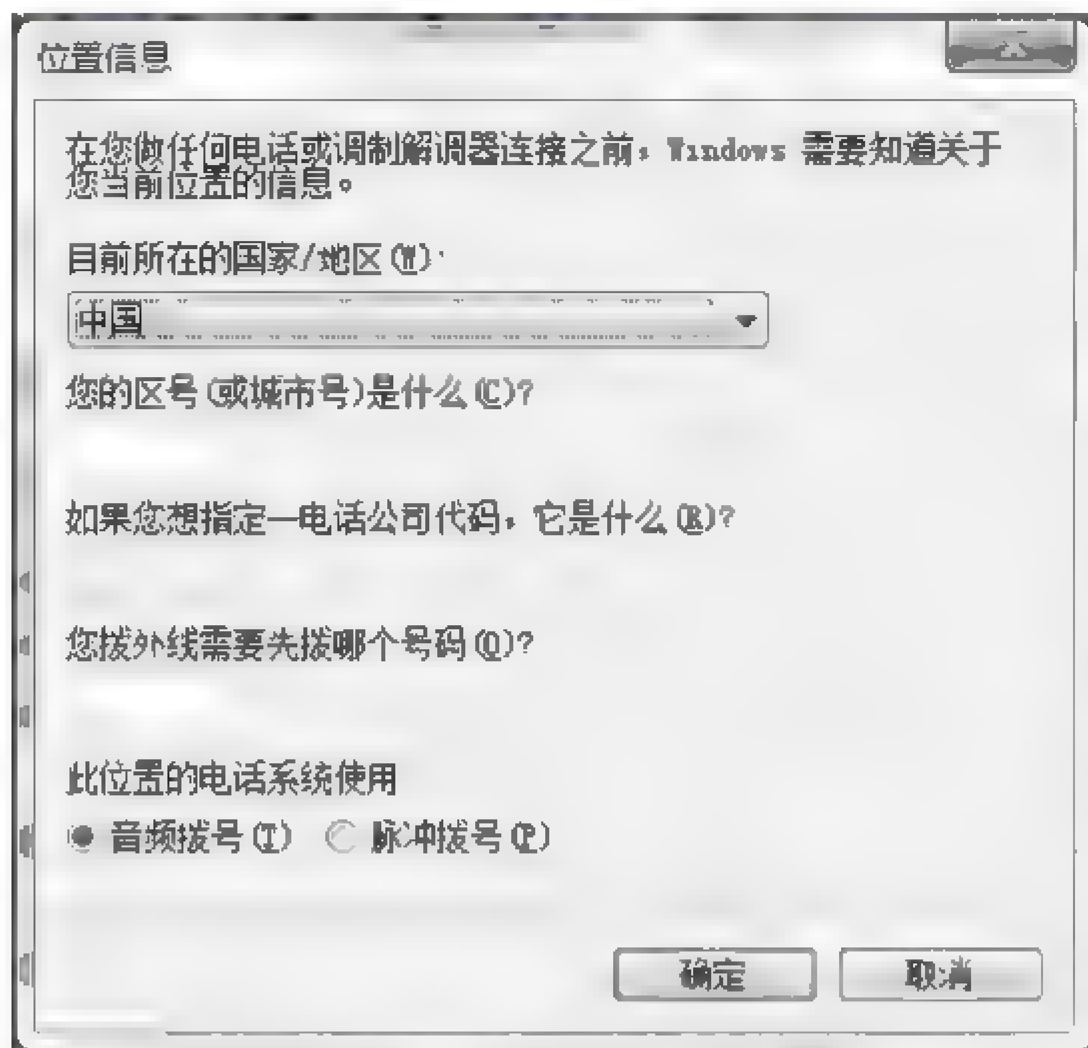


图 6-16　配置截图 1



图 6-17　配置截图 2

输入名称，单击“确定”按钮进入下一步，如图 6-18 所示。

选择连接串口，这里选中 IVT 启动的串口，然后单击“确定”按钮，进入下一个配置界面，如图 6-19 所示。



图 6-18　配置截图 3



图 6-19　配置截图 4

这个界面中只需要修改“位/秒”选项，选中“115200”，然后单击“确定”按钮完成创建，此时超级终端的界面如图 6-20 所示。

当蓝牙端需要模拟发送手机报文的功能时，其设置方法如下：

首先在“新建的连接”界面的菜单中找到“文件→属性”，然后在出现的窗口中，选中“设置”标签页中的“ASCII 码设置”按钮，如图 6-21 所示。

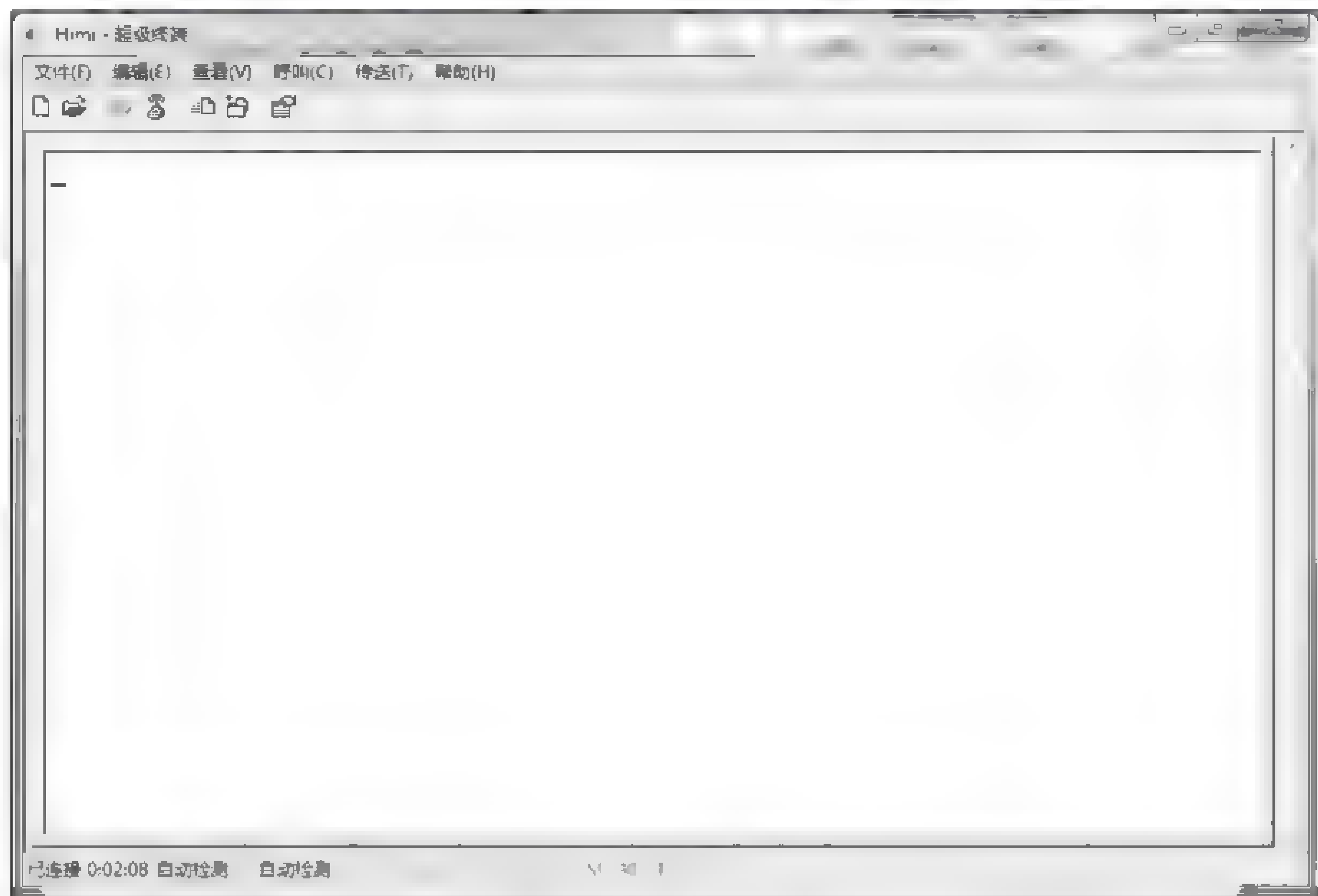


图 6-20　配置截图 5

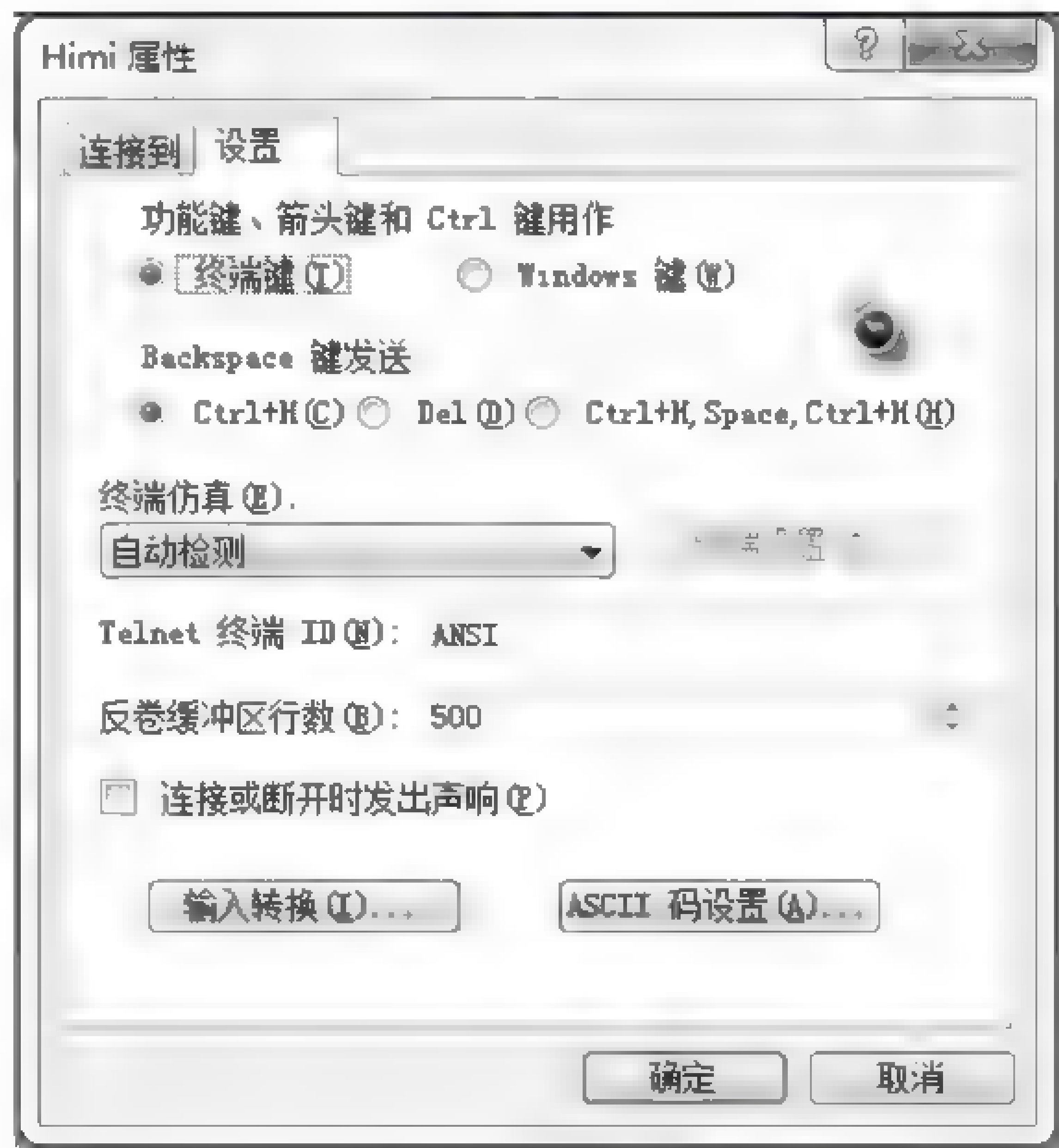


图 6-21　配置截图 6

然后在出现的设置界面中，选中如图 6-22 所示的圈起的选项。

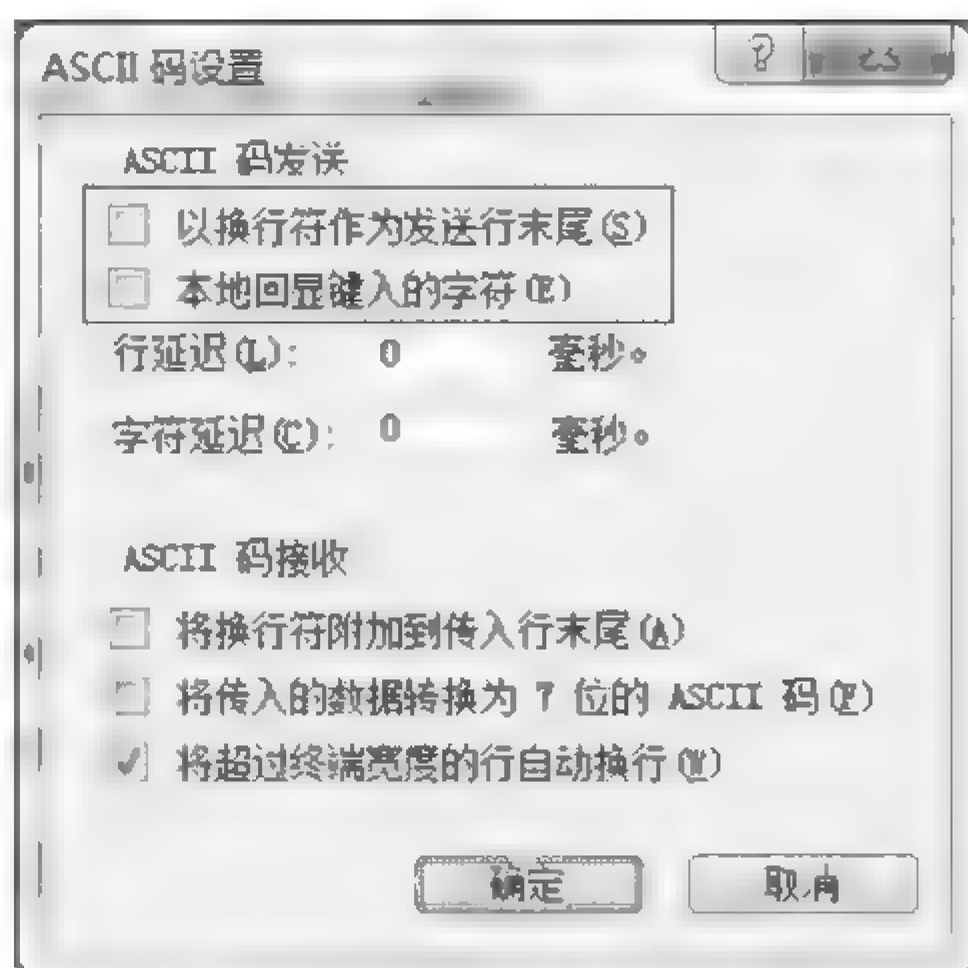


图 6-22 配置截图 7

最后一直单击“确定”按钮即可完成蓝牙的设置。此时可以通过键盘输入文字，输入的字符默认会发送给配对连接的手机端。

蓝牙开发的准备工作做完之后，接下来看一个范例。新建项目“BlueToothProject”，游戏框架为 SurfaceView 游戏框架，项目对应的源代码为“6-9（蓝牙对战游戏）”。

这个项目实现一个蓝牙对战游戏。当正常配对连接到 IVT（其他蓝牙设备）时进入游戏，游戏中存在两个填充圆形，红色圆形表示当前设备，蓝色圆形表示连接的其他蓝牙设备。当前设备通过触屏操控圆形的移动，而其他设备是通过 IVT 软件模拟的蓝牙终端，所以需要通过电脑键盘的“w、s、a、t”4 个按键来操控。

由于本项目代码很长，在这里不再详细讲解每行代码，只对重点的代码段进行对应的备注。对于游戏的绘制与组件的添加等内容，在之前的章节中都详细讲解过，这里也不再讲解。

首先在 AndroidManifest.xml 中添加蓝牙权限：

```
<!-- 声明蓝牙权限 -->
<uses-permission android:name="android.permission.BLUETOOTH" />
<!-- 允许程序发现和配对蓝牙设备 -->
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
      在 strings.xml 中定义组件的名字：
      <string name="openBlueTooth">打开蓝牙</string>
      <string name="blueToothIsVisible">蓝牙可见</string>
      <string name="searchDrives">搜索设备</string>
      <string name="connectDrives">连接设备</string>
      <string name="choiceConnectDrives">请选择连接设备</string>
```

因为本 Demo 是将游戏视图与组件同时显示，所以修改 Main.xml 布局文件：

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical" android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <RelativeLayout android:layout_width="fill_parent"
                android:layout_height="wrap_content"
                android:layout_weight="1">
        <com.himi.MySurfaceView
                android:layout_width="fill_parent"
                android:layout_height="fill_parent" />
        <Button android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_alignParentBottom="true"
                android:text="@string/openBlueTooth"
                android:id="@+id/Btn_OpenBt" />
        <Button android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_alignParentBottom="true"
                android:layout_toRightOf="@id/Btn_OpenBt"
                android:text="@string/blueToothIsVisible"
                android:id="@+id/Btn_BtIsVisible" />
        <Button android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_alignParentBottom="true"
                android:layout_toRightOf="@id/Btn_BtIsVisible"
                android:text="@string/searchDrives"
                android:id="@+id/Btn_SearchDrives" />
        <Button android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_alignParentBottom="true"
                android:layout_toRightOf="@id/Btn_SearchDrives"
                android:text="@string/connectDrives"
                android:id="@+id/Btn_ConnectDrives" />
        <TextView android:id="@+id/textview"
                android:layout_width="fill_parent"
                android:layout_height="fill_parent" android:text="
                                      蓝牙对战 Demo 示例"
                android:textSize="32sp" android:textColor="#000000"
                android:gravity="center_horizontal" />
    </RelativeLayout>
</LinearLayout>
```

此布局文件的可视化效果如图 6-23 所示。



图 6-23 main.xml 布局效果

MainActivity 是主 Activity 类，其代码如下：

```
public class MainActivity extends Activity implements OnClickListener {
      //按钮组件
      public static Button btConnect, btOpen, btIsVisible, btSearch;
      //蓝牙适配器
      public static BluetoothAdapter btAda;
      //UUID 协议
      public static final String SPP_UUID = "00001101-0000-1000-8000-
00805F9B34FB";
      //蓝牙连接
      public static BluetoothSocket btSocket;
      //单利本类
      public static MainActivity ma;
      @Override
      public void onCreate(Bundle savedInstanceState) {
             super.onCreate(savedInstanceState);
             ma = this;
             //这里没有设置全屏，为了便于观察当前蓝牙是否打开的状态变化
              //getWindow().setFlags(WindowManager.LayoutParams.
              FLAG_FULLSCREEN, WindowManager.LayoutParams.FLAG_FULLSCREEN);
             this.requestWindowFeature(Window.FEATURE_NO_TITLE);
             setContentView(R.layout.main);
             //实例按钮
             btOpen = (Button) findViewById(R.id.Btn_OpenBt);
             btIsVisible = (Button) findViewById(R.id.Btn_BtIsVisible);
             btSearch = (Button) findViewById(R.id.Btn_SearchDrives);
             btConnect = (Button) findViewById(R.id.Btn_ConnectDrives);
             //为按钮绑定监听器
             btOpen.setOnClickListener(this);
```

```
        btIsVisible.setOnClickListener(this);
        btSearch.setOnClickListener(this);
        btConnect.setOnClickListener(this);
        //实例蓝牙适配器
        btAda = BluetoothAdapter.getDefaultAdapter();
        if (btAda.getState() == BluetoothAdapter.STATE_OFF) {
            btOpen.setText("打开蓝牙");
        } else if (btAda.getState() == BluetoothAdapter.STATE_ON) {
            btOpen.setText("关闭蓝牙");
        }
        // 注册 Receiver 来获取蓝牙设备相关的结果
        IntentFilter intent = new IntentFilter();
        // 远程设备发现动作。
        intent.addAction(BluetoothDevice.ACTION_FOUND);
        //远程设备的键态的变化动作。
        intent.addAction(BluetoothDevice.ACTION_BOND_STATE_CHANGED);
        //蓝牙扫描本地适配器模改变动作。
        intent.addAction(BluetoothAdapter.ACTION_SCAN_MODE_CHANGED);
        //状态改变动作
        intent.addAction(BluetoothAdapter.ACTION_STATE_CHANGED);
        registerReceiver(searchDevices, intent);//注册接收
    }
    //监听动作
    private BroadcastReceiver searchDevices = new BroadcastReceiver() {
        public void onReceive(Context context, Intent intent) {
            String action = intent.getAction();
            //搜索设备时，取得设备的 MAC 地址
            if (BluetoothDevice.ACTION_FOUND.equals(action)) {
                BluetoothDevice device = intent.
                getParcelableExtra(BluetoothDevice.EXTRA_DEVICE);
                String str = "设备：" + device.getName() + "*" +
                        device.getAddress();
                if (MySurfaceView.vc_str != null) {
                    if (MySurfaceView.vc_str.size() != 0) {
                        for (int j = 0; j<MySurfaceView.
                                vc_str.size(); j++) {
                            // 防止重复添加
                            if (MySurfaceView.vc_str.
                                elementAt(j).equals(str)
                                 == false) {
                            // 容器添加发现的设备名称和 mac 地址
                            MySurfaceView.vc_str.
                              addElement(str);
                            }
                        }
                    } else {
```

```
        MySurfaceView.vc_str.addElement(str);
                            }
                        }
                    }
                }
    };

    @Override
    protected void onDestroy() {
        this.unregisterReceiver(searchDevices);
        super.onDestroy();
        System.exit(0);
    }
    @Override
    public void onClick(View v) {
        if (MySurfaceView.gameState != MySurfaceView.CONNTCTED) {
            if (v == btOpen) {//蓝牙开关
                if (btAda.getState() == BluetoothAdapter.
                    STATE_OFF) {
                      btAda.enable();
                      btOpen.setText("关闭蓝牙");
                } else if (btAda.getState() == BluetoothAdapter.
                          STATE_ON) {
                      btAda.disable();
                      btOpen.setText("打开蓝牙");
                }
            } else if (v == btIsVisible) {//蓝牙是否可见
                Intent intent = new Intent(BluetoothAdapter.
                        ACTION_REQUEST_DISCOVERABLE);
        intent.putExtra(BluetoothAdapter.EXTRA_DISCOVERABLE_DURATION,
                    110);
                //第二个参数是本机蓝牙被发现的时间，系统默认范围[1-300],
                //超过范围默认300，小于范围默认120
                startActivity(intent);
            } else if (v == btSearch) {//搜索蓝牙
                // 如果蓝牙还没打开
                if (btAda.getState() == BluetoothAdapter.
                    STATE_OFF){Toast.makeText(MainActivity.this,
                      "请先打开蓝牙", 1000).show();
                  return;
                }
                setTitle("本机蓝牙地址: " + btAda.getAddress());
                MySurfaceView.vc_str.removeAllElements();
                btAda.startDiscovery();
            } else if (v == btConnect) {
                if (MySurfaceView.vc_str.size() == 0) {
```

```
                    Toast.makeText(MainActivity.this, "当前没有设备",
                    1000).show();
                    } else {
                        Intent intent = new Intent();
                         //打开显示搜索的蓝牙设备的Activity
                         intent.setClass(this,ChoiceDrivesList.class);
                        this.startActivity(intent);
                    }
                }
            } else {
               Toast.makeText(this, "这里只是一个Demo示例，很多情况没有进行处
                    理，为了等出现误操作造成异常，请重新运行项目！", Toast.
                    LENGTH_LONG).show();
               this.finish();
            }
        }
}
```

ChoiceDrivesList类用于显示搜索到的所有蓝牙设备。

```
public class ChoiceDrivesList extends Activity {
        //所有蓝牙设备的名字
        private String[] names;
        //提示
        private Toast toast;
        //对话框显示当前搜索到的蓝牙设备
        private AlertDialog.Builder dialog;

        public ChoiceDrivesList() {
                names = new String[MySurfaceView.vc_str.size()];
                for (int i = 0; i < MySurfaceView.vc_str.size(); i++) {
                        names[i] = MySurfaceView.vc_str.elementAt(i);
                }
        }
        public void DisplayToast(String str, int type) {
                try {
                        toast = null;
                        if (type == 0) {
                                toast = Toast.makeText(this, str,
                                        Toast.LENGTH_SHORT);
                        } else {
                                toast = Toast.makeText(this, str,
                                        Toast.LENGTH_LONG);
                        }
                        toast.setGravity(Gravity.TOP, 0, 220);
                        toast.show();
                } catch (Exception e) {
```

```
                // TODO: handle exception
            }
        }
        @Override
        public void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);
                dialog = new AlertDialog.Builder(ChoiceDrivesList.this);
                dialog.setIcon(android.R.drawable.btn_dialog);
                dialog.setSingleChoiceItems(names, 0, new DialogInterface.
                        OnClickListener() {
                        public void onClick(DialogInterface dialog, int which) {
                                MySurfaceView.deviceIndex = which;
                        }
                        }).setIcon(R.drawable.icon).setPositiveButton("连接", new
                        DialogInterface.OnClickListener() {
                        public void onClick(DialogInterface dialoginterface, int i) {
                            DisplayToast("正在连接设备: " +
                            MySurfaceView.vc_str.elementAt(MySurfaceView.
                            deviceIndex), 1);
                             MySurfaceView.gameState = MySurfaceView.CONNTCTING;
                             new ConnectThread().start();
                             finish();
                        }
                }).setNegativeButton("取消", new DialogInterface.
                        OnClickListener() {
                        public void onClick(DialogInterface dialoginterface, int i) {
                                finish();
                        }
                }).setTitle("请选择连接设备!");
                dialog.show();
        }
}
```

由于本类是一个 Activity 活动，所以在 AndroidManifest.xml 文件中需要声明以下内容：

```
<activity android:name=".ChoiceDrivesList"
            android:label="@string/choiceConnectDrives"
            android:configChanges="orientation|keyboardHidden"
            android:screenOrientation="portrait"
            android:theme="@android:style/Theme.Dialog">
</activity>
```

ConnectThread 类的主要作用是，当成功连接其他蓝牙设备后，线程不断接受报文数据，其代码如下：

```
public class ConnectThread extends Thread {
```

```
    private BlueToothServer bts;
    @Override
    public void run() {
        //取消可见
        MainActivity.btAda.cancelDiscovery();
        String str =
MySurfaceView.vc_str.elementAt(MySurfaceView.deviceIndex);
        //这里 split 的参数是采用正则表达式规则
        String[] values = str.split("\\*");
        //split 此方法把字符串分割成多个字符串,这里分成两个字符串
        // *号 之前一个, *号 之后 一个
        String address = values[1];//这里就是取出连接设备的 mac 地址
        //Android 蓝牙普遍支持 SPP 协议
        UUID uuid = UUID.fromString(MainActivity.SPP_UUID);
        //实例蓝牙设备
        BluetoothDevice btDevice = MainActivity.btAda.
        getRemoteDevice(address);
        try {
            MainActivity.btSocket = btDevice.
                createRfcommSocketToServiceRecord(uuid);
            MainActivity.btSocket.connect();
        } catch (IOException e) {
            Log.e("Himi", "Connected Error!");
            Toast.makeText(MainActivity.ma, "无法连接此设备!",
                        1000).show();
            e.printStackTrace();
            return;
        }
        //当正常连接配对其他蓝牙设备后启动线程，一直监听报文数据
        MySurfaceView.gameState = MySurfaceView.CONNTCTED;
        bts = new BlueToothServer();
        bts.flag = true;
        bts.start();
    }
}
class BlueToothServer extends Thread {
    private InputStream ips;
    boolean flag;
    @Override
    public void run() {
        while (flag) {
            if (MySurfaceView.gameState == MySurfaceView.CONNTCTED)
{
                try {
                    //没有接收到数据，这里一直处于阻塞状态
                    ips = MainActivity.btSocket.
```

```
                                        getInputStream();
                                byte[] buffer = new byte[1024];
                                if (ips.read(buffer) != -1) {
                                        String str=new String(buffer, 0, 1);
                                        if (str.equals("w")) {//上
                                                MySurfaceView.other_Arcy -= 5;
                                        } else if (str.equals("s")) {//下
                                                MySurfaceView.other_Arcy += 5;
                                        } else if (str.equals("a")) {//左
                                                MySurfaceView.other_Arcx -= 5;
                                        } else if (str.equals("d")) {//右
                                                MySurfaceView.other_Arcx += 5;
                                        }
                                }
                        } catch (IOException e) {
                                Log.e("Himi", "inPutStream is Error!!");
                                e.printStackTrace();
                        }
                }
            }
        }
}
```

最后是整个游戏视图 MySurfaceView 类：

```
public class MySurfaceView extends SurfaceView implements Callback,
Runnable {
        private Thread th;
        private SurfaceHolder sfh;
        private Canvas canvas;
        private Paint paint;
        private boolean flag;
        //用于存储搜索到的蓝牙设备
        public static Vector<String> vc_str;
        //未连接蓝牙设备
        public static final int NONE = 1;
        //正在连接蓝牙设备
        public static final int CONNTCTING = 2;
        //已连接蓝牙设备
        public static final int CONNTCTED = 3;
        //当前蓝牙连接状态
        public static int gameState = NONE;
        //连接蓝牙设备的下标索引
        public static int deviceIndex;
        public static int myArc_x = 50, myArc_y = 150, other_Arcx = 110,
other_Arcy = 150;
        private OutputStream ops;
```

```
public MySurfaceView(Context context, AttributeSet attrs) {
        super(context, attrs);
        vc_str = new Vector<String>();
        this.setKeepScreenOn(true);
        sfh = this.getHolder();
        sfh.addCallback(this);
        paint = new Paint();
        paint.setAntiAlias(true);
        this.setLongClickable(true);
        this.setFocusable(true);
        this.setFocusableInTouchMode(true);
        Log.e("Himi", "surfaceChanged");
}
public void surfaceCreated(SurfaceHolder holder) {
        flag = true;
        th = new Thread(this, "himi_Thread_one");
        th.start();
        Log.e("Himi", "surfaceCreated");
}
public void surfaceChanged(SurfaceHolder holder, int format, int
                            width, int height) {
        Log.e("Himi", "surfaceChanged");
}
public void surfaceDestroyed(SurfaceHolder holder) {
        flag = false;
        Log.e("Himi", "surfaceDestroyed");
}
public void myDraw() {
        try {
                canvas = sfh.lockCanvas();
                if (canvas != null) {
                        canvas.drawColor(Color.WHITE);
                        paint.setColor(Color.RED);
                        switch (gameState) {
                        case NONE:
                                if (vc_str != null) {
                                        for (int i=0;i<vc_str.size();i++){
                                                paint.setTextSize(12);
                         canvas.drawText(vc_str.elementAt(i), 3, 150 +
                                i * 30, paint);
                                        }
                                }
                                break;
                        case CONNTCTING:
                                paint.setTextSize(20);
                        canvas.drawText("正在连接设备:", 3, 150, paint);
```

```
                    paint.setTextSize(12);
                    canvas.drawText(vc_str.elementAt(deviceIndex),3,
                            190, paint);
                        break;
                    case CONNTCTED:
                        paint.setTextSize(20);
                        paint.setTextSize(12);
                        canvas.drawText("已成功连接:" +
                        vc_str.elementAt(deviceIndex), 10, 110,
                                paint);
                        paint.setColor(Color.RED);
                        canvas.drawCircle(myArc_x, myArc_y, 20,
                                 paint);
                        paint.setColor(Color.BLUE);
                        canvas.drawCircle(other_Arcx, other_Arcy,
                                  20, paint);
                        paint.setColor(Color.BLACK);
                        canvas.drawText("我方圆形", myArc_x - 20,
                                myArc_y - 25, paint);
                        canvas.drawText("对方圆形", other_Arcx - 20,
                               other_Arcy - 25, paint);
                        break;
                    }
                }
            } catch (Exception e) {
                Log.v("Himi", "draw is Error!");
                e.printStackTrace();
            } finally {
                if (canvas != null)
                    sfh.unlockCanvasAndPost(canvas);
            }
    }
    @Override
    public boolean onKeyDown(int keyCode, KeyEvent event) {
            return true;
    }
    @Override
    public boolean onTouchEvent(MotionEvent event) {
            if (gameState == CONNTCTED) {
                try {
                    ops = MainActivity.btSocket.getOutputStream();
                    byte bx[] = null;
                    byte by[] = null;
                    myArc_x = (int) event.getX();
                    myArc_y = (int) event.getY();
                    bx = new String("X " + myArc_x).getBytes();
```

```
                    by = new String("X " + myArc_x).getBytes();
                    if (bx != null && by != null) {
                        ops.write(bx);
                        ops.write(by);
                    }
                } catch (IOException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
            return true;
        }
        private void logic() {
        }
        public void run() {
            while (flag) {
                logic();
                myDraw();
                try {
                    Thread.sleep(100);
                } catch (Exception ex) {
                }
            }
        }
}
```

项目运行效果如图 6-24 所示。



图 6-24 项目效果图

当 IVT 模拟蓝牙终端发送报文数据时，其界面如图 6-25 所示。

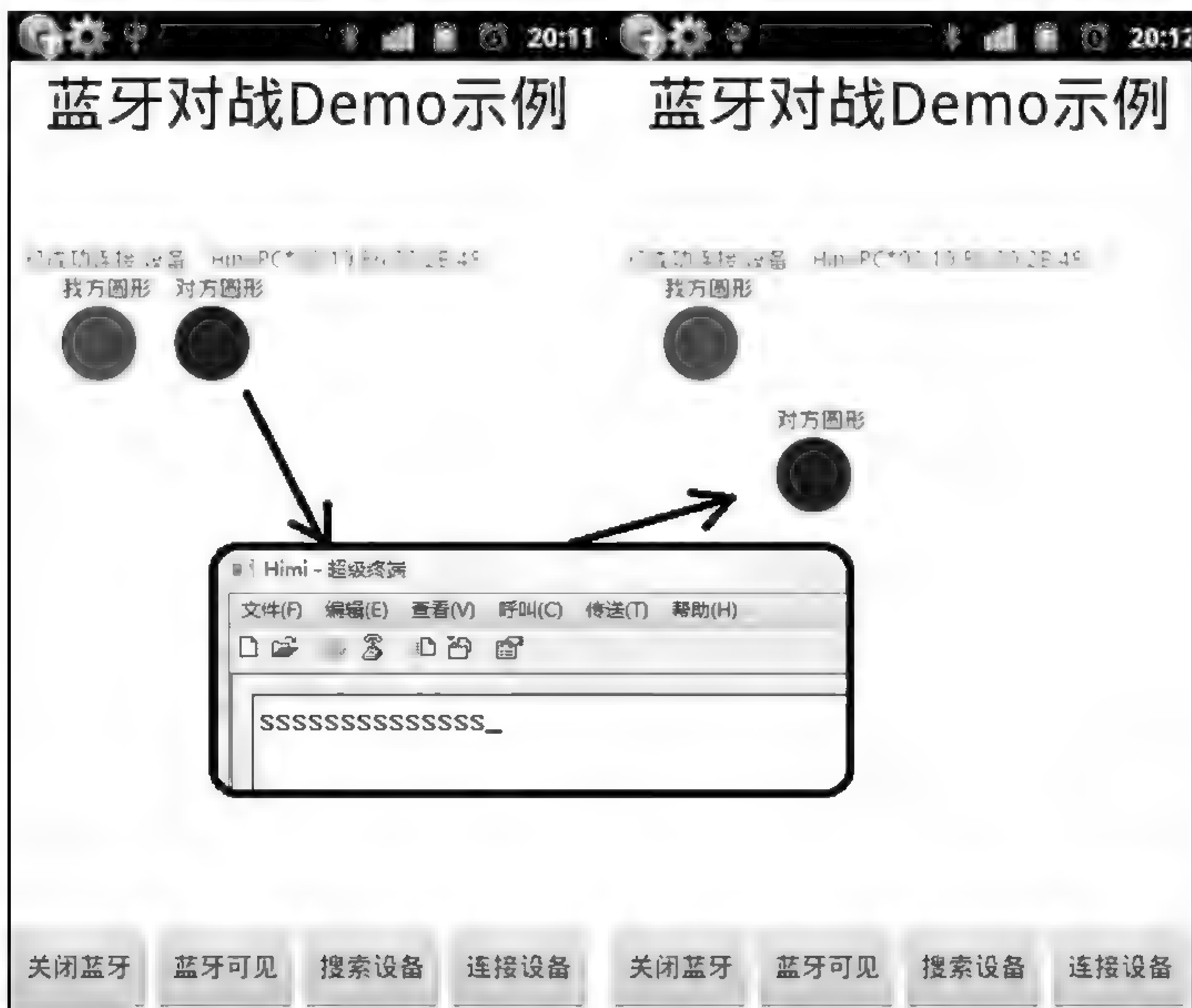


图 6-25　IVT 终端控制圆形移动

当前设备发送给 IVT 蓝牙终端报文数据时，其界面如图 6-26 所示。

当前设备操控红色圆形移动，也会将圆形的坐标发送给配对的蓝牙设备，这里是发给 IVT 模拟的蓝牙终端。

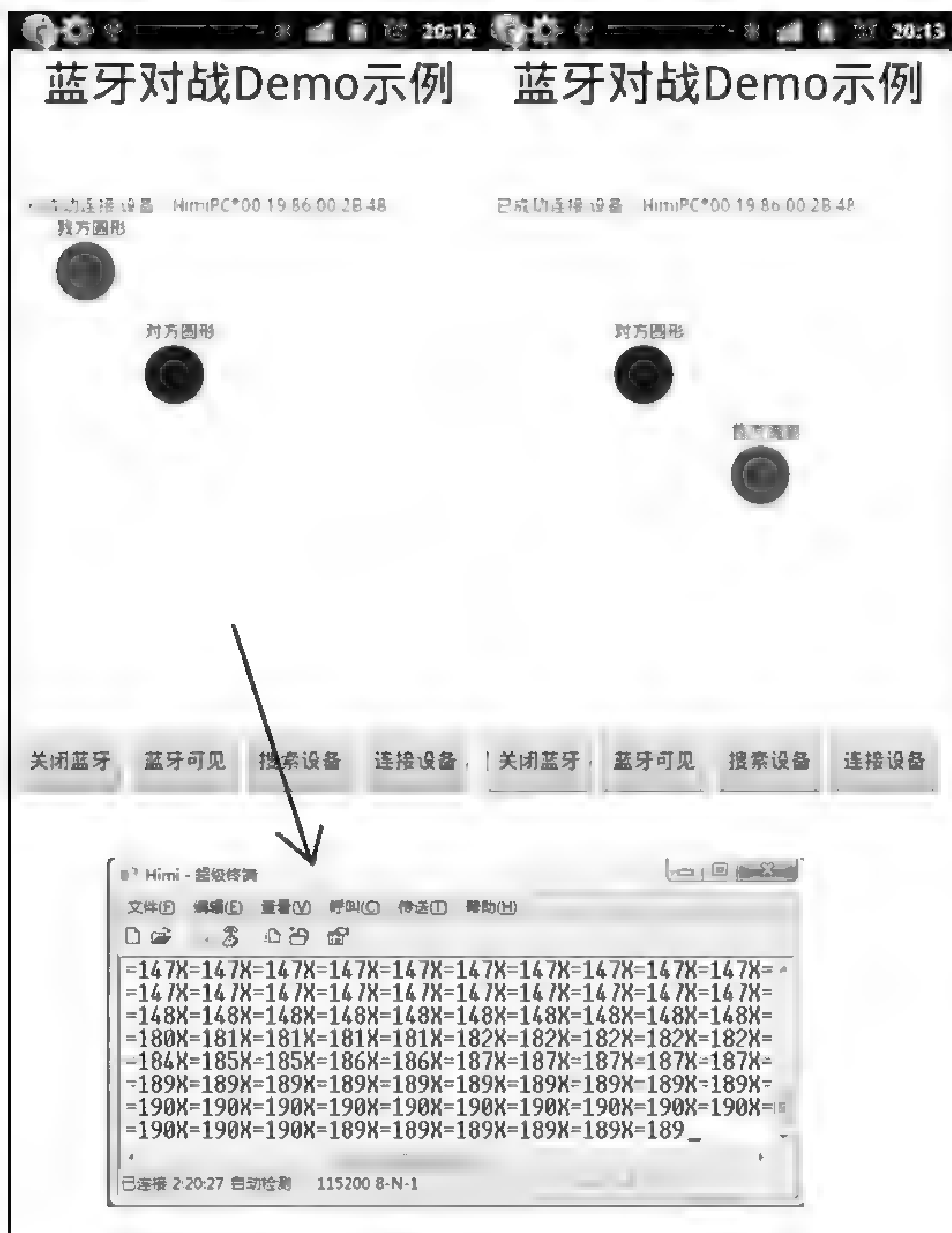


图 6-26　当前设备控制圆形移动

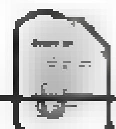
6.10 网络游戏开发基础

在手机网游开发中，客户端与服务端的通信方式经常使用两种协议，一种是 Socket 协议，另外一种是 Http 协议。两种协议最主要的区别在于：

Socket 协议属于长连接，一旦客户端正常连接到服务器端，如果两端没有单纯的断开操作，那么下次交互数据就不需要再次连接，两者将一直维持交互状态。这种交互协议适用于即时通信类型的游戏，例如 ARPG、RPG 等。

Http 协议属于短连接，当客户端正常连接到服务端后，一旦数据交互完后就会断开，不会像 Socket 那样维持连接状态。

对于这两种通信协议，这里也是简单地描述了一下，详细的说明可以参阅其他书籍和资料。



注意

本小节服务端是由 Java 项目实现的。

6.10.1 Socket

通信数据都是以流进行读写，那么应该优先考虑如何得到输入输出流。

```
InputStream is = Socket.getInputStream();
OutputStream os =Socket.getOutputStream();
```

客户端与服务器获取输入输出流的方法都是通过 Socket 实例来实现，然后进行通信，最后对数据进行读写操作。

Socket类客户端的构造方法：

```
Socket  socket  = Socket(String dstName, int dstPort);
```

- 第一个参数：服务器地址；
- 第二个参数：服务器端口。

Socket类服务端的构造方法：

```
Socket  socket  = ServerSocket.accept();
```

其作用是监听是否有客户端连接，当客户端正常连接后会返回一个 Socket 实例。ServerSocket类是通过端口实例化的。

下面简单完成客户端（Android 项目）发送给服务端（Java 项目）一字符串，然后服务器接受并返回此字符串给客户端。本小节对应源代码为“6-10-1（Socket 协议）”。

1．创建服务端

新建 Java 项目“MySocketServer”，新建类 MyServer，其代码如下：

```
public class MyServer {
      //服务器连接
      public static ServerSocket serverSocket;
      //连接
      public static Socket socket;
      //端口
      public static final int PORT = 8888;
      public static void main(String[] args) {
```

```
            DataInputStream dis = null;
            DataOutputStream dos = null;
            try {
                serverSocket = new ServerSocket(PORT);
                System.out.println("正在等待客户端连接...");
                //这里处于等待状态，如果没有客户端连接，程序不会向下执行
                socket = serverSocket.accept();
                dis = new DataInputStream(socket.getInputStream());
                dos = new DataOutputStream(socket.getOutputStream());
                //读取数据
                String clientStr = dis.readUTF();
                //写出数据
                dos.writeUTF(clientStr);
                System.out.println("----客户端已成功连接!----");
                //得到客户端的 IP
                System.out.println("客户端的 IP =" +
                  socket.getInetAddress());
                //得到客户端的端口号
                System.out.println("客户端的端口号 =" + socket.getPort());
                //得到本地端口号
                System.out.println("本地服务器端口号=" +
                   socket.getLocalPort());
                System.out.println("-----------------------");
                System.out.println("客户端: " + clientStr);
            } catch (IOException e) {
                e.printStackTrace();
            } finally {                 try {
                    if (dis != null)
                        dis.close();
                    if (dos != null)
                        dos.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
}
```

代码很简单，服务端主要是监听端口，一旦客户端连接上，就先读取数据，然后再写给客户端。

2 实现客户端

新建项目“SocketClient”，创建主 Activity 类 MainActivity。因为通信需要权限，所以需要在 AndroidManifest.xml 文件中声明联网权限：

```
<uses-permission android:name "android.permission.INTERNET" />
```

然后简单写一个布局，添加一个“发送”按钮、文本编辑组件等。

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
      android:orientation="vertical" android:layout_width="fill_parent"
      android:layout_height="fill_parent">
      <TextView android:layout_width="fill_parent"
      android:layout_height="wrap_content" android:text="@string/hello" />
 <EditText android:id="@+id/edit"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content" />
<Button android:id="@+id/Btn_commit"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content" android:text="@string/send" />
<TextView android:layout_width="fill_parent" android:id="@+id/tv"
      android:layout_height="wrap_content" android:text="@string/get" />
</LinearLayout>
```

布局效果如图 6-27 所示。



图 6-27　布局示意图

在 strings.xml 文件中添加 3 个字符串：

```
<string name "hello">这里输入文字发给服务器</string>
<string name "send">发送</string>
<string name "get">这里显示服务器发来的信息!</string>
```

最后修改 MainActivity 主活动类，代码如下：

```
public class MainActivity extends Activity implements OnClickListener {
	private Button btn_ok;
	private EditText edit;
	private TextView tv;
	//Socket 用于连接服务器获取输入输出流
	private Socket socket;
	//服务器 server/IP 地址
	private final String ADDRESS = "192.168.1.100";
	//服务器端口
	private final int PORT = 8888;
	@Override
	public void onCreate(Bundle savedInstanceState) {
		super.onCreate(savedInstanceState);

	this.getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCR
				EEN, WindowManager.LayoutParams.FLAG_FULLSCREEN);
		this.requestWindowFeature(Window.FEATURE_NO_TITLE);
		setContentView(R.layout.main);
		btn_ok = (Button) findViewById(R.id.Btn_commit);
		tv = (TextView) findViewById(R.id.tv);
		edit = (EditText) findViewById(R.id.edit);
		btn_ok.setOnClickListener(this);
	}
	public void onClick(View v) {
		if (v == btn_ok) {
			DataInputStream dis = null;
			DataOutputStream dos = null;
			try {
				//阻塞函数，正常连接后才会向下继续执行
				socket = new Socket(ADDRESS, PORT);
				dis = new DataInputStream(socket.
					getInputStream());
				dos = new DataOutputStream(socket.
					getOutputStream());
				//向服务器写数据
				dos.writeUTF(edit.getText().toString());
				String temp = "I say:";
				temp += edit.getText().toString();
				temp += "\n";
				temp += "Server say:";
				//读取服务器发来的数据
				temp += dis.readUTF();
				tv.setText(temp);
			} catch (IOException e) {
```

```
                    Log.e("Himi", "Stream error!");
                    e.printStackTrace();
                } finally {
                    try {
                        if (dis != null)
                            dis.close();
                        if (dos != null)
                            dos.close();
                    } catch (IOException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                    }
                }
            }
        }
}
```

这里要提醒的一点：因为服务端是先获取客户端发来的数据，然后再写给客户端，所以客户端应该是先写给服务端数据，然后再读取服务端发来的数据，如图 6-28 所示。



图 6-28　写入读取顺序对比

首先运行服务器端，运行效果如图 6-29 所示。



图 6-29　启动服务端

然后运行客户端，并点击发送数据，如图 6-30 所示。

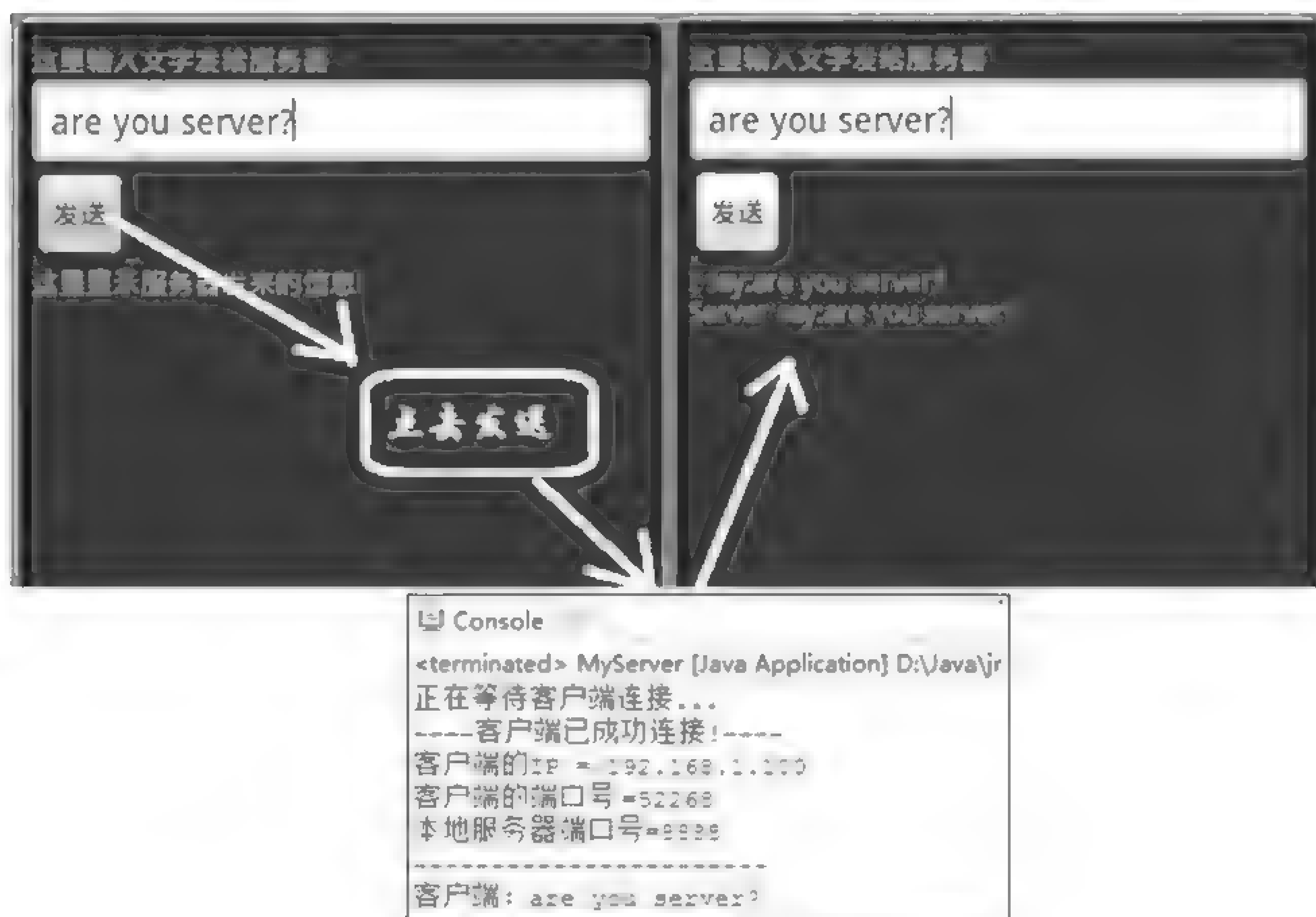


图 6-30　Client 与 Server 通信

此实例只是简单实现客户端与服务端的一次性通信。其实服务端监听端口事件应该放在线程中不断地去监听是否有客户端连接（毕竟网游中不只一个客户端），然后为每个客户端分配一个线程去处理读写数据事件。本小节只是让读者了解和熟悉客户端与服务端是如何连接与通信的，由于网游服务与客户端的细节处理过多，这里就不再详述，更详细的交互与细节处理可以参阅其他书籍和资料。

6.10.2 Http

利用 Http 通讯同样需要分为服务端与客户端。由于服务端需要使用 Java 项目实现，牵扯到 J2EE 的知识，所以这里就不再利用 Java 实现服务端，而是简单地使用百度网站作为服务端。

首先优先考虑如何得到输入输出流：

```
URLConnection urlConnect = URL.openConnection()
```

URLConnection 类通过地址 URL 连接服务器，并获取连接实例 URLConnection，然后通过 URLConnection 类获取输入输出流，用于读取和写入数据操作：

```
//获取输入流
DataInputStream dis = new DataInputStream
(urlConnect.getInputStream());
//获取输出流
```

```
DataOutputStream dos = new DataOutputStream(urlConnect.getOutputStream());
```

接下来通过 Http 协议访问“百度”获取数据的示例，来熟悉 Android 客户端是如何连接到服务端，并且获取输入输出流对数据进行操作的。

新建项目“MyHttpClient”，主 Activity 类为 MainActivity，项目对应的源代码为“6-10-2（Http 协议）”。首先在 AndroidManifest.xml 中添加联网权限：

```
<uses-permission android:name="android.permission.INTERNET" />
```

然后修改 main.xml 布局，添加一个按钮和文本组件：

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
      android:orientation="vertical" android:layout_width="fill_parent"
      android:layout_height="fill_parent">
      <LinearLayout android:orientation="vertical"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent">
<Button android:id="@+id/Btn_commit"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="连接服务器" />
<TextView android:layout_width="fill_parent"
      android:id="@+id/tv"
      android:layout_height="wrap_content"
      android:text="显示获取到的服务器信息" />
      </LinearLayout>
</ScrollView>
```

由于获取的数据可能在屏幕中无法完整显示，所以整个布局使用了 ScrollView 滚动显示。布局如图 6-31 所示。



图 6-31　布局示意图

然后修改MainActivity类：

```
public class MainActivity extends Activity implements OnClickListener {
    private Button btn_ok;
    private TextView tv;
    private final String ADDRESS = "http://www.baidu.com";
    //声明http连接
    private URLConnection urlConnect;
    //声明服务器地址
    private URL url;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

    this.getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCR
                EEN, WindowManager.LayoutParams.FLAG_FULLSCREEN);
        this.requestWindowFeature(Window.FEATURE_NO_TITLE);
        setContentView(R.layout.main);
        btn_ok = (Button) findViewById(R.id.Btn_commit);
        tv = (TextView) findViewById(R.id.tv);
        btn_ok.setOnClickListener(this);
    }
    public void onClick(View v) {
        DataInputStream dis = null;
        if (v == btn_ok) {
            try {
                //实例地址
                url = new URL(ADDRESS);
                //实例Http连接
                urlConnect = url.openConnection();
                //获取输入流
                dis = new DataInputStream(urlConnect.
                                getInputStream());
                //获取输出流
                //DataOutputStream dos = new DataOutputStream
                            (urlConnect.getOutputStream());
                //获取服务器返回的数据
                int temp = 0;
                ByteArrayBuffer baff = new ByteArrayBuffer(1000);
                while ((temp = dis.read()) != -1) {
                    baff.append(temp);
                }
                //将服务器返回的信息显示在文本

    tv.setText(EncodingUtils.getString(baff.toByteArray(),
            "UFT-8"));
```

```
                    } catch (MalformedURLException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                    } catch (IOException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                    } finally {
                        try {
                            if (dis != null)
                                dis.close();
                        } catch (IOException e) {
                            // TODO Auto-generated catch block
                            e.printStackTrace();
                        }
                    }
                }
        }
}
```

项目运行效果如图 6-32 所示。

其实在 Android 中，Http 访问服务器的方式很多，例如也可以利 HttpURLConnection 来实现连接 Http 协议访问服务端。



图 6-32　Http 访问服务器

6.11 本地化与国际化

一款游戏或者一个应用开发完成后，可能会放到国外的平台渠道；不同地区使用的语言是不相同的，比如中国大陆使用简体中文，台湾地区使用繁体中文等。那么一款简体中文的Android 应用，如果投放在国外或者中国台湾地区是不是要重新替换所有字符，重新打包呢？答案是否定的。

其实做法很简单，在之前的章节介绍 Android 目录结构时，曾经提到过在 res 目录下的drawable 目录，从 SDK 1.6 以后就变成了 drawble-hdpi、drawable-ldpi、drawable-mdpi 3 个目录，其作用是 Android 在运行程序时，会根据当前设备信息选择对应的资源文件夹。而Android 项目资源文件 res 目录下的 values 目录，在运行应用时，会根据当前不同的手机设备语言选择不同的 values 路径；当然 values 的其他语言的版本，ADT 并没有自动生成，需要自定义文件目录。

下面通过实例来详细讲解这个多语言功能的实现方法。新建项目“LocalProject”，游戏框架为 SurfaceView 游戏框架，项目对应的源代码为“6-11（本地化与国际化）”。

首先在 string.xml 中添加一个字符串：

```
<string name="localTest">LocalTest</string>
```

修改 MySurfaceView 类的绘图函数，绘制出新添加的字符串：

```
public void myDraw() {
      ...
      canvas.drawText(this.getResources().getString(R.string.
      localTest), 40, 40, paint);
      ...
}
```

运行项目，效果如图 6-33 所示。

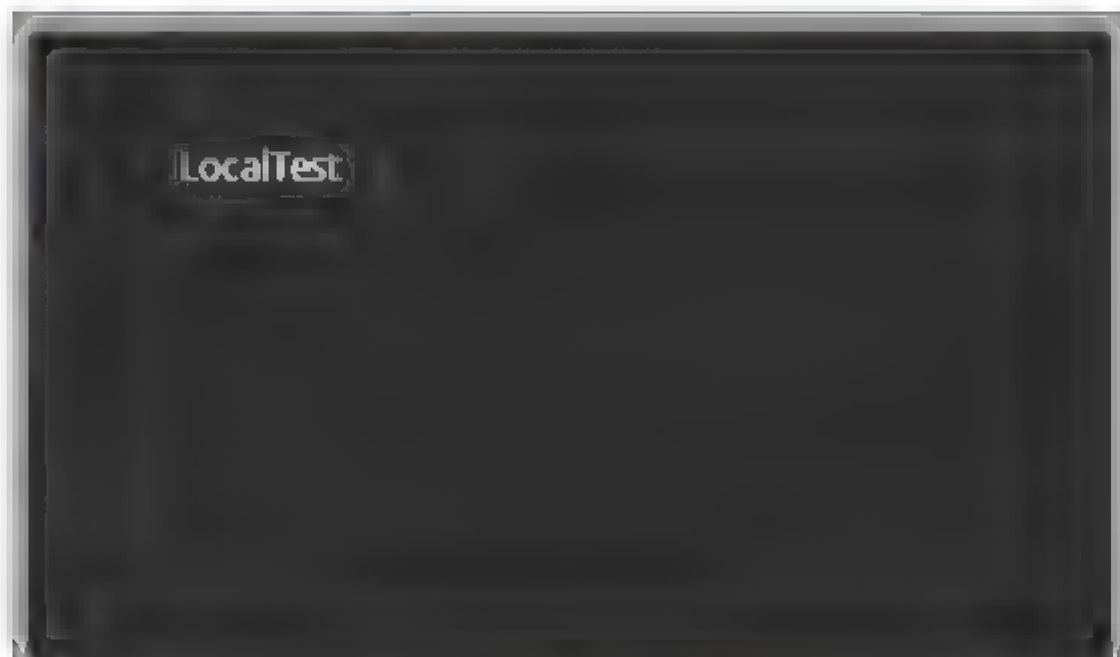


图 6-33 效果图 1

下面在 res 目录下添加 3 个新的 values 目录，values-en-rUS、values-zh-rCN 和 values-zh-rTW，如图 6-34 所示。

目录	内容
values strings.xml	<string name="localTest">LocalTest</string>
values-en-rUS strings.xml	<string name="localTest">(English):LocalTest</string>
values-zh-rCN strings.xml	<string name="localTest">(中文简体语言):LocalTest</string>
values-zh-rTW strings.xml	<string name="localTest">(中文繁體語言):LocalTest</string>

图 6-34 values 不同语言版本

目录的命名格式为“values-语言缩写-r 国家地区简写”，比如：

- values-en-rUS :对应设备语言是英文;
- values-zh-rCN: 对应设备语言是中文简体;
- values-zh-rTW: 对应设备语言是中文繁体。

为了更好地看出区别与效果，每个不同语言版本的 values 中定义的 localTest 值，都设置不一样的字符串。设置模拟器或者设备的语言为英文，并且重新运行项目，效果如图 6-35 所示。

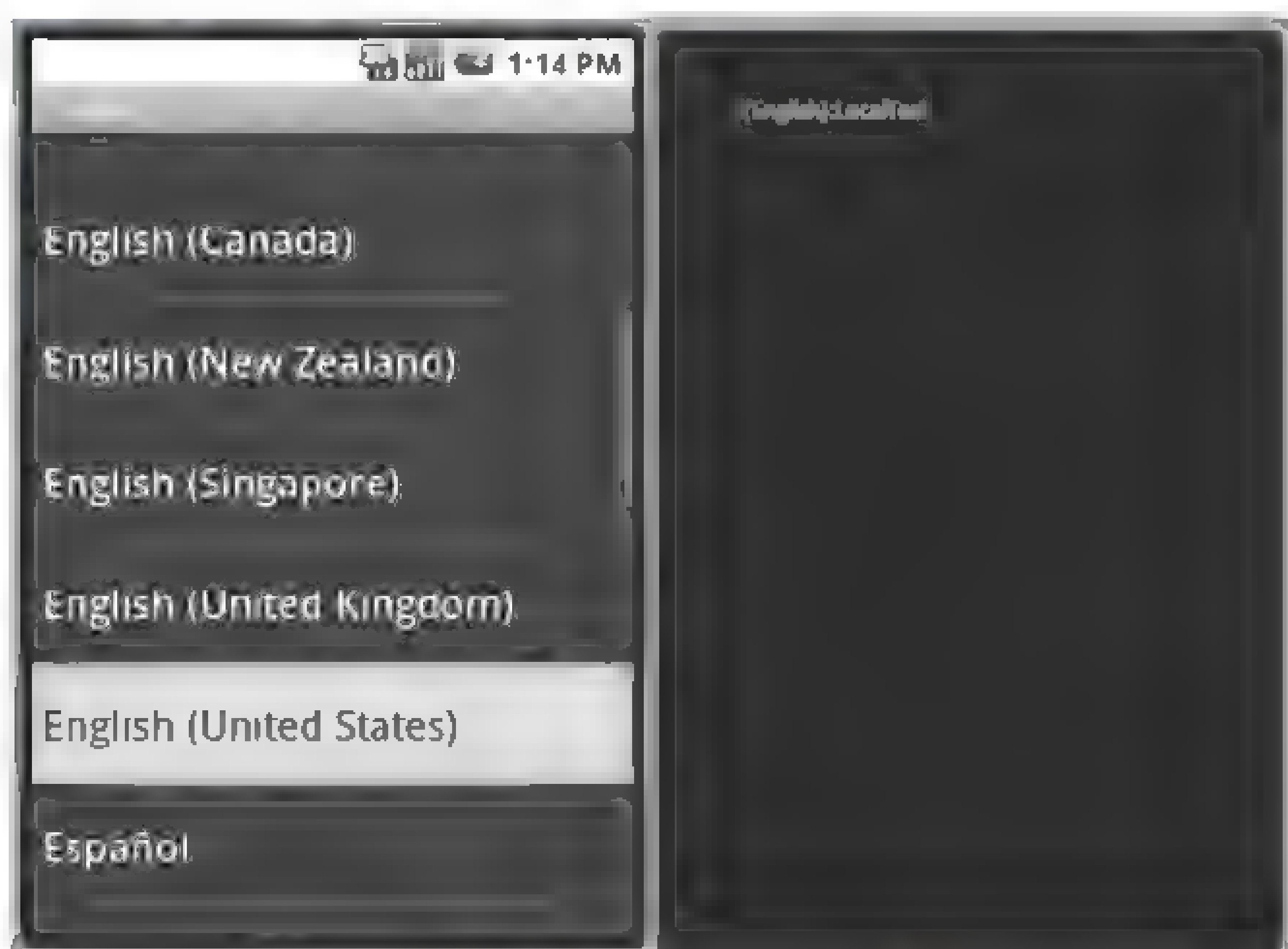


图 6-35 英文版项目截图

设置模拟器或者设备的语言为中文简体，并且重新运行项目，效果如图 6-36 所示。

设置模拟器或者设备的语言为中文繁体，并且重新运行项目，效果如图 6-37 所示。

当设备的语言在项目中找不到对应语言版本的 values 目录时，默认使用 values 路径。设置模拟器或者设备的语言为法语，并且重新运行项目，效果如图 6-38 所示。



图 6-36　中文简体版项目截图



图 6-37　中文繁体版项目截图

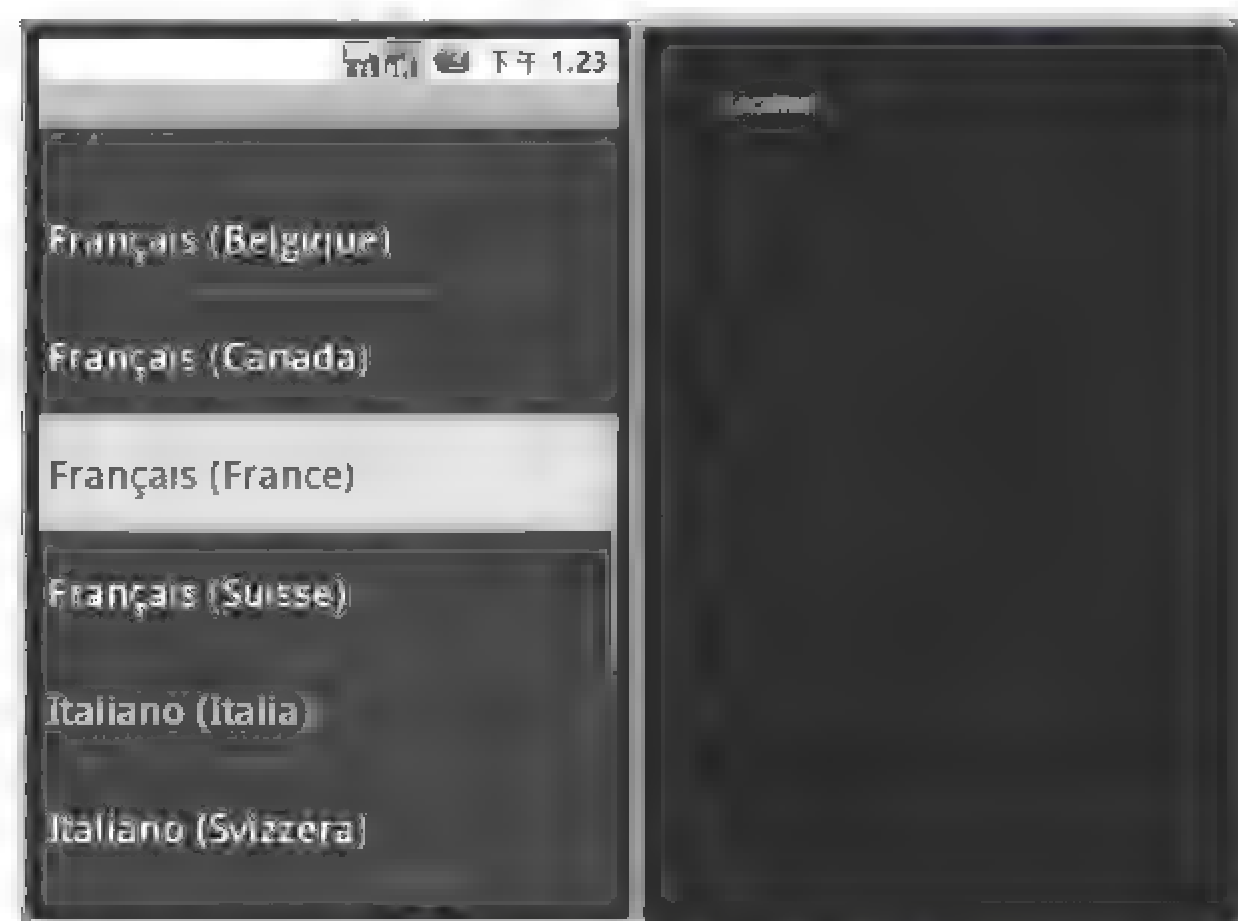


图 6-38　法语版本项目截图

当然，有时需要强制使用一种语言版本，可以通过代码进行设置：

```
Configuration conf = new Configuration();//实例配置信息
conf.locale = Locale.TAIWAN;//修改本地化语言
```

当配置文件设置完毕后，还需要更新当前项目的配置即可：

```
this.getResources().updateConfiguration(conf, null);//更新
```

重新设置模拟器或者设备的语言为非繁体中文，并且重新运行项目，效果如图 6-39 所示。

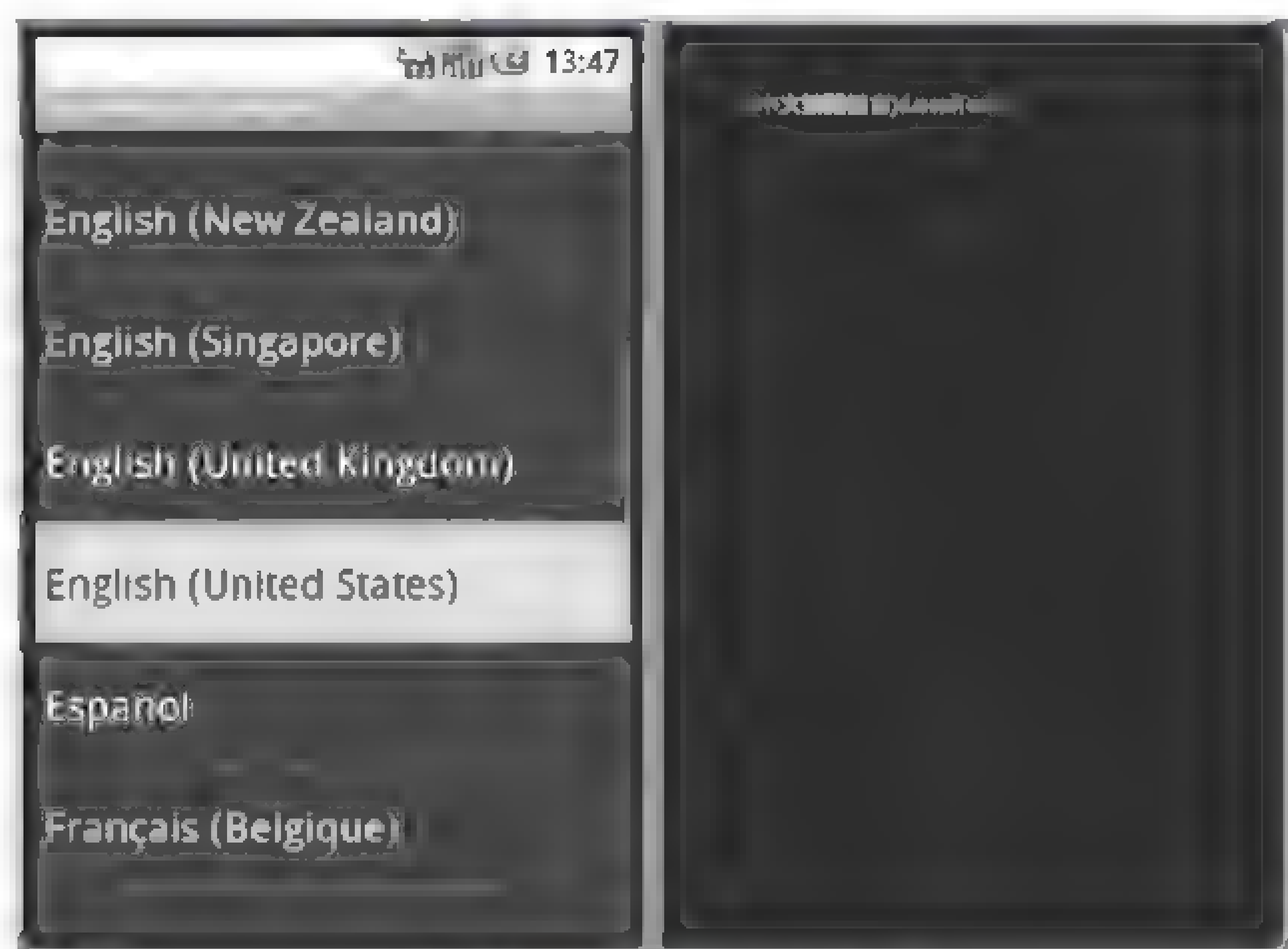


图 6-39　强制语言版本

从图 6-39 中可以看到，将设备语言选择英文，运行项目因强制设置了台湾语言，所以项目索引的 values 是 values-zh-rTW，显示为中文繁体。强制语言版本只是针对当前运行的项目生效，不会影响整个设备的显示语言。

需要注意的是，在多语言应用开发中，还经常用到一个方法Locale.getDefault()，这个方法的作用是获取当前默认语言区域。

通过本小节的讲解，读者可以更好地体会利用 xml 定义 Android 应用多语言所带来的方便。

6.12 本章小结

本章介绍了一个游戏开发的高级知识，内容包括 360° 平滑游戏导航摇杆、多触点实现图片缩放、触屏手势识别、加速度传感器、9patch 工具的使用、代码实现截屏功能、效率检视工具、游戏视图与系统组件共同显示、蓝牙对战游戏、网络游戏开发基础、本地化与国际化等，这些内容都是游戏开发人员必须掌握的。

第 7 章

Box2D 物理引擎

从本章节可以学习到：

- Box2D 概述
- 将 Box2D 类库导入 Android 项目中
- 物理世界与手机屏幕坐标系之间的关系
- 创建 Box2D 物理世界
- 创建矩形物体
- 让物体在屏幕中展现
- 创建自定义多边形物体
- 物理世界中的物体角度
- 创建圆形物体
- 多个 Body 的数据赋值
- 设置 Body 坐标与给 Body 施加力
- Body 碰撞监听、筛选与 Body 传感器
- 关节
- 通过 AABB 获取 Body
- 物体与关节的销毁



7.1 Box2D 概述

Box2D 是一款用于 2D 游戏的物理引擎。在 Box2D 物理世界里，创建出的每个物体都更接近于真实世界中的物体，让游戏中的物体运动起来更加真实可信，让游戏世界看起来更具交互性。

Android 平台常见的十几款游戏引擎中，例如：Rokon、AndEngine、libgdx 等物理引擎都封装了 Box2D 物理引擎，可见 Box2D 在物理引擎中占据了多重要的位置。

Box2D 在很多平台都有对应的版本：Flash 版本、Iphone 版本、Java 版本（JBox2D）等等，在本书中开发 Android 语言采用的是 Java，所以这里介绍的对应 Box2D 平台也是 Java 平台，称为 JBox2D，对应的版本为 JBox2d 2.0。

7.2 将 Box2D 类库导入 Android 项目中

添加 Box2D 类库到 Android 项目详细步骤如下：

步骤1 新建项目“HelloBox2d”（创建 Android 项目与之前创建项目步骤无差异）。

步骤2 然后在项目里添加一个“lib”目录用于存放 Box2D 类库（“.jar”）。

步骤3 右键项目“HelloBox2d”，选中“New”→“Folder”选项，然后出现“New Folder”窗口，如图 7-1 所示。



图 7-1　New Folder

图 7-1 所示的窗口中，在“Folder name”文本框中填写目录名，这个目录名可以自定义，一般导入外部类库目录都使用 lib 与 libs 来命名，然后单击“Finish”按钮完成目录的创建。

此时只是将 Box2D 提供的类库放入 Android 项目中，但是并没有添加到项目的类库中，所以还需要将 Box2D 包添加到项目类库中！

步骤3 右击项目“HelloBox2d”选中“Properties ”→“Java Build Path”→“Libraries”→“Add JARs...”，选中“HelloBox2d”项目下 lib 目录下的 Jbox2d 类库的 jar 包，最后一直单击“OK”按钮返回即可，如图 7-2、图 7-3 所示。

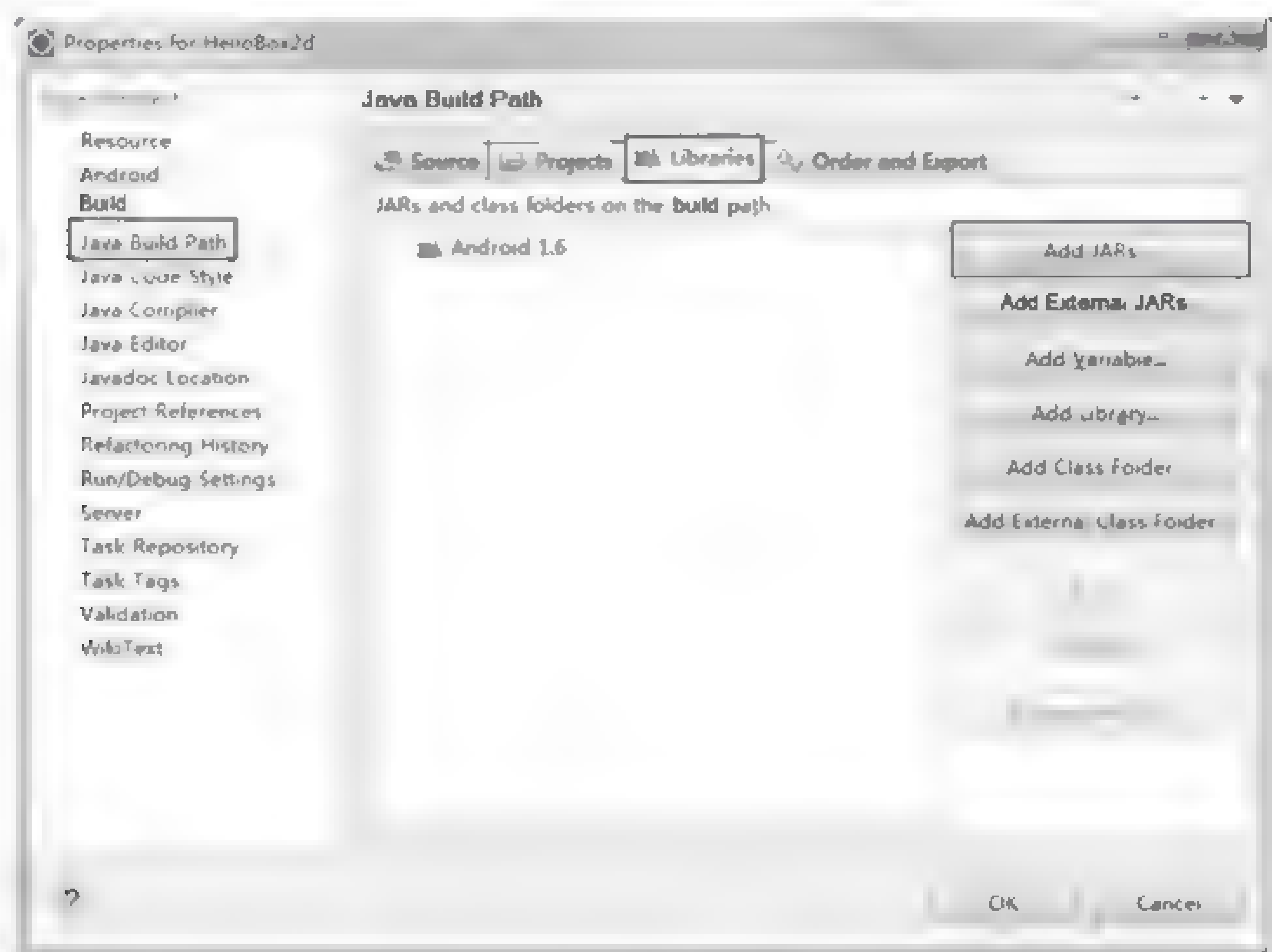


图 7-2　项目配置

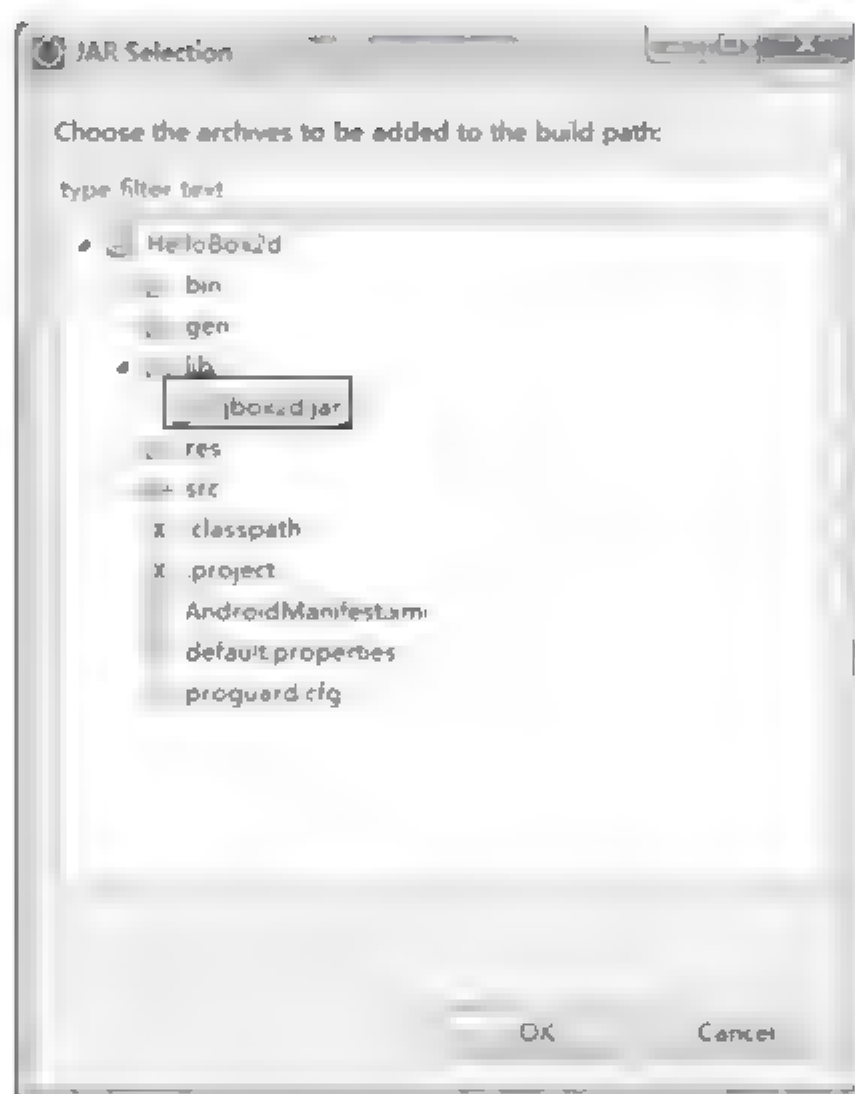


图 7-3　项目中添加 Box2d 类库

到此，在 Android 项目中添加 JBox2D 物理引擎类库的步骤就结束了，现在就开始进入 Box2D 引擎的物理世界吧！

7.3 物理世界与手机屏幕坐标系之间的关系

本节来讲解一下物理世界与手机屏幕坐标系之间的关系。假设创建一个 200 米的物理世界，然后观察其物理世界与手机屏幕之间的坐标系关系，如图 7-4 所示。

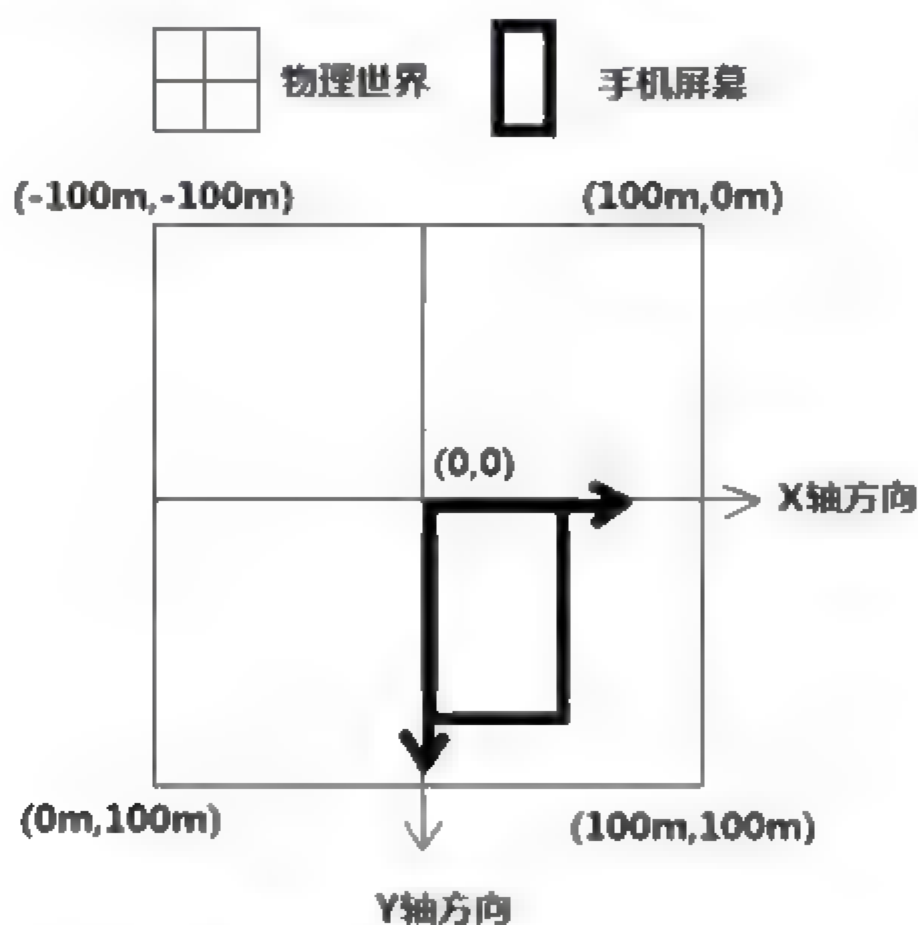


图 7-4　物理世界与手机屏幕坐标系关系图

从图 7-4 中可以很清晰的看出，手机屏幕的左上角（0,0）坐标，正是物理世界的中心点坐标；手机屏幕绘制图形时，一般默认以左上角作为锚点！而在 Box2d 的物理世界中，一个新的 Body（物体）等被创建出来之后，默认以其质心（可以近似为中心点）作为锚点；如图 7-5 所示，是“在屏幕上绘制一张图片，并且在物理世界中添加一个物体”的位置关系图。

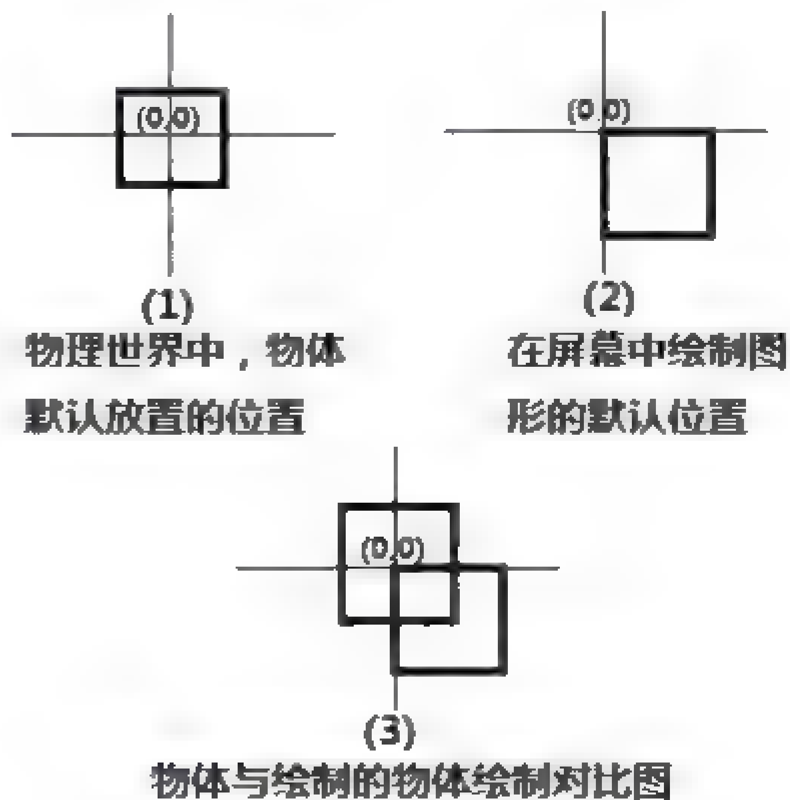


图 7-5　默认放置的位置对比图

除此之外，Box2D为了使物体与关节等更加贴切的模拟现实，在Box2D引擎中使用的长度单位是“米（m）”，所以Box2D引擎中的一些方法的长度参数不再是以像素为单位，而是需要转换成“米”；反之，从Box2D引擎函数返回值中得到的长度值也是以“米”做单位的，使用其值前需要将其转换为像素，然后再使用。

关于米与像素之间的换算关系，其实是通过自定义一个现实与屏幕的比例关系进行换算的，后续章节会有讲解。

7.4 创建Box2D物理世界

World（jbox2d.dynamics.World）类就是Box2D引擎中的物理世界。不管是Body（物体）还是Joint（关节）都必须放在Box2D这个物理世界中，因为物理世界也是有范围的，一旦物体和关节不在世界范围内，它们将不会进行物理模拟。

下面就来创建一个物理世界，范例项目对应的源代码为“7-4（Box2d物理世界）”。

首先看一下World类的构造函数：

```
World(AABB world AABB,Vec2 gravity,boolean doSleep)
```

World类只有这一种构造方式，它的三个参数的含义如下：

- 第一个参数：AABB类的实例，AABB表示一个物理模拟世界的范围；
- 第二个参数：Vec2实例，一个二维世界向量类，在Box2D中的最常用的一种数据类型；在这里表示物理世界的重力方向；
- 第三个参数：布尔值，表示在物理世界中，如果静止不动的物体是否对其进行休眠。如果设置其值为“true”，则表示当物理世界开始进行模拟时，在这个物理世界中静止没有运行的物体都将进行休眠，除非物体被施加了力的作用或者与其他物体发生碰撞之后会被唤醒；如果设置其值设置为“false”，那么物理世界中的所有物体不管是否静止都会一直进行物理模拟。

创建一个物理世界代码如下：

```
AABB aabb = new AABB();// 实例化物理世界的范围对象
aabb.lowerBound.set(-100, -100);// 设置物理世界范围的左上角坐标
aabb.upperBound.set(100, 100);// 设置物理世界范围的右下角坐标
Vec2 gravity = new Vec2(0, 10);// 实例化物理世界重力向量对象
World world = new World(aabb, gravity, true);// 实例化物理世界对象
```

以上代码中有两点需要注意：

（1）aabb设置物理世界范围传入的参数，不要理解成像素！在Box2d的物理世界中，被

认为是现实生活中的“米（m）”单位。

（2）设置物理世界的重力向量（gravity），其两个参数在这里分别表示物理世界中的 X 轴与 Y 轴方向上的重力数值，其值的“+”“-”号在这里表示 X 与 Y 轴的重力方向，X 轴正值表示向右，Y 轴正值表示向下；因为是模拟真实世界，所以这里的 X 重力向量设置为零，Y 轴方向设置为现实生活中的重力值：10（可以理解为 10N）。

刚才的一段代码就已经创建了一个物理世界，但只是定义了物理世界，并没有开始进行物理模拟，所以还需要 world 设置物理模拟：

```
world.step(float timeStep, int iterations);
```

此函数表示让物理世界开始进行物理模拟，其两个参数含义如下：

- 第一个参数：表示（时间步）物理世界模拟的频率；
- 第二个参数：表示（迭代值）迭代值越大模拟越精确，但性能越低。

这里要注意以下几点：

①因为物理世界模拟具有持续性，所以应该将设置放在线程中，不断的让物理世界进行模拟。

②时间步：应该与游戏的刷新率相同，否则物理世界模拟将不同步。

③迭代值：可以理解为在单次时间步中进行遍历模拟运算数据的次数。

④在 Box2D 中最常使用的单位是 float 浮点数类型，作者刚接触 Box2D 时，在定义物理世界模拟频率时，写成了以下错误的形式：

```
float timeStep = 1 / 60;
```

这样写导致物理世界的物体永远不运动，其原因就是“1/60”的值永远是零！所以正确书写形式应该是：

```
float timeStep = 1f / 60f;
```

到此一个物理世界真正的创建出来并且进行模拟了，但是因为物理世界中并没有放置任何的物体，所以运行项目在视觉中将看不到任何的效果，下面的章节中将开始在物理世界中创建物体。

作者推荐物理模拟的频率一般设为每秒 60 帧，迭代设为 10，具体设置根据应用和设备性能情况而定。

在后续创建物体和关节的章节中，很多代码需要传入以“米”作为单位的数值，所以为了便于转换，可以定义一个成员变量：

```
final float RATE = 30;
```

在 Box2D 的物理世界中，为了更加贴切的模拟现实，部分函数参数不再使用“像素”而

是用“米”表示。为了能将模拟的物理世界映射到手机屏幕中，定义一个屏幕与现实世界的比例变量“RATE”，这个比例值作者推荐设置为 30，因为一般不会修改此值，所以可以定义为 final 常量类型。

还要注意定义此变量不要用 int 类型，应该用 float 类型。否则会发生如同 timeStep 类似的状况，此值可能会比预计的小。例如：

```
int RATE = 30  ->  45/RATE = 1
float RATE = 30 -> 43/RATE = 1.5
```

7.5 创建矩形物体

在学习创建矩形物体之前，首先要理解几个基本概念：

- 大家可以想象一下现实生活中的物体基本上都是由圆形与多边形组成，所以在 Box2d 物理世界中存在两种 2D 图形，一种是圆形，一种是多边形。
- 在 Box2D 中物体的创建都应该设置质量、摩擦力与恢复力这三个基本属性。
- Box2D 属于工厂模式，也就是说在 Box2D 的物理世界中创建物体，都是由工厂（World）生成的，而不是 new 出来的。

World 创建一个物体的步骤则分为以下三步：

步骤1 首先创建物体皮肤。

步骤2 然后创建物体刚体。

步骤3 最后通过皮肤与刚体信息去创建一个物体。

简单熟悉了 Box2D 创建一个物体的步骤后，下面来创建一个多边形，添加在物理世界中，项目对应的源代码为“7-5（在物理世界中添加多边形）”。

在项目中添加一个 createPolygon 函数，代码如下：

```java
public Body createPolygon(float x, float y, float width, float height,
        boolean isStatic) {
        // ---创建多边形皮肤
        PolygonDef pd = new PolygonDef(); // 实例一个多边形的皮肤
        if (isStatic) {
                pd.density = 0; // 设置多边形为静态
        } else {
                pd.density = 1; // 设置多边形的质量
        }
        pd.friction = 0.8f; // 设置多边形的摩擦力
        pd.restitution = 0.3f; // 设置多边形的恢复力
```

```
        // 设置多边形快捷成盒子(矩形)
        // 两个参数为多边形宽高的一半
        pd.setAsBox(width / 2 / RATE, height / 2 / RATE);
        // ---创建刚体
        BodyDef bd = new BodyDef(); // 实例一个刚体对象
        // 设置刚体的坐标
        bd.position.set((x + width / 2) / RATE, (y + height / 2) / RATE);
        // ---创建 Body (物体)
        Body body = world.createBody(bd); // 物理世界创建物体
        body.createShape(pd); // 为 Body 添加皮肤
        body.setMassFromShapes(); // 将整个物体计算打包
        return body;
}
```

以上代码中，各个属性的含义说明如下：

- 质量（density）：当物体质量设置为 0 时，此物体视为“静态物体”；所谓“静态物体”表示不需要运动的物体；比如现实生活中的山、房门等这些没有外力不会发生运动的物体则认为是静态不运动的。
- 摩擦力（friction）：取值通常设置在 0~1 之间，0 意味着没有摩擦，1 会产生最强摩擦。
- 恢复力（restitution）：取值也通常设置在 0 ~ 1 之间，0 表示物体没有恢复力，1 表示物体拥有最大恢复力。
- 刚体设置坐标的时候，需要传入现实生活中的“米”做为参数单位，所以这里除以比例“RATE”，将像素单位转换为“米”。
- BodyDef.position.set（float x, float y）方法，设置 Body 相对于物理世界的坐标。

在此之前已经介绍过，物理世界中创建出的物体默认放置的位置是以物理中心点为锚点，那么为了让其与手机屏幕绘制图形位置重合，需要将其物理的 X 位置加上其宽的一半，其物体的 Y 位置加上其高的一半，这样就相当于将其 Body 的锚点设置成了左上角，如图 7-6 所示。

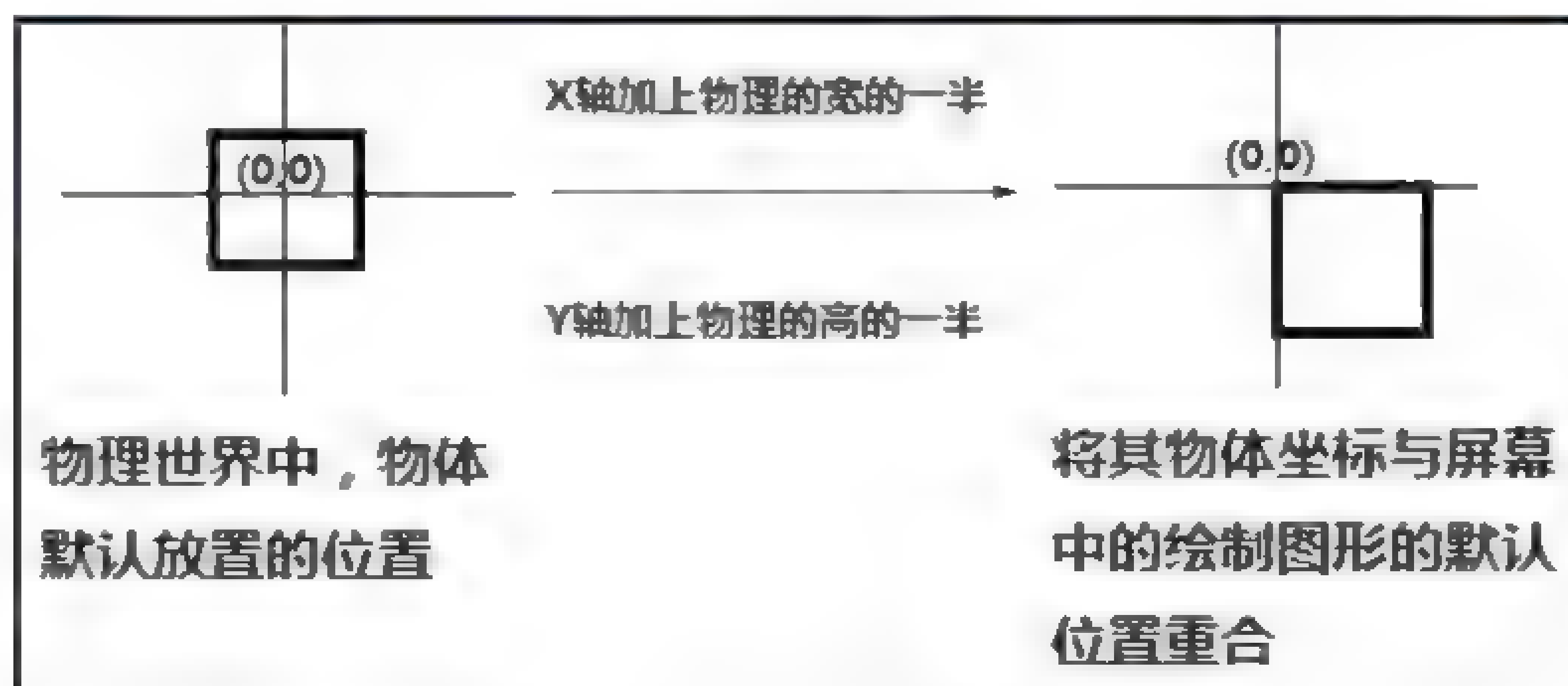


图 7-6　物体与图形坐标重合

项目运行效果如图7-7所示。

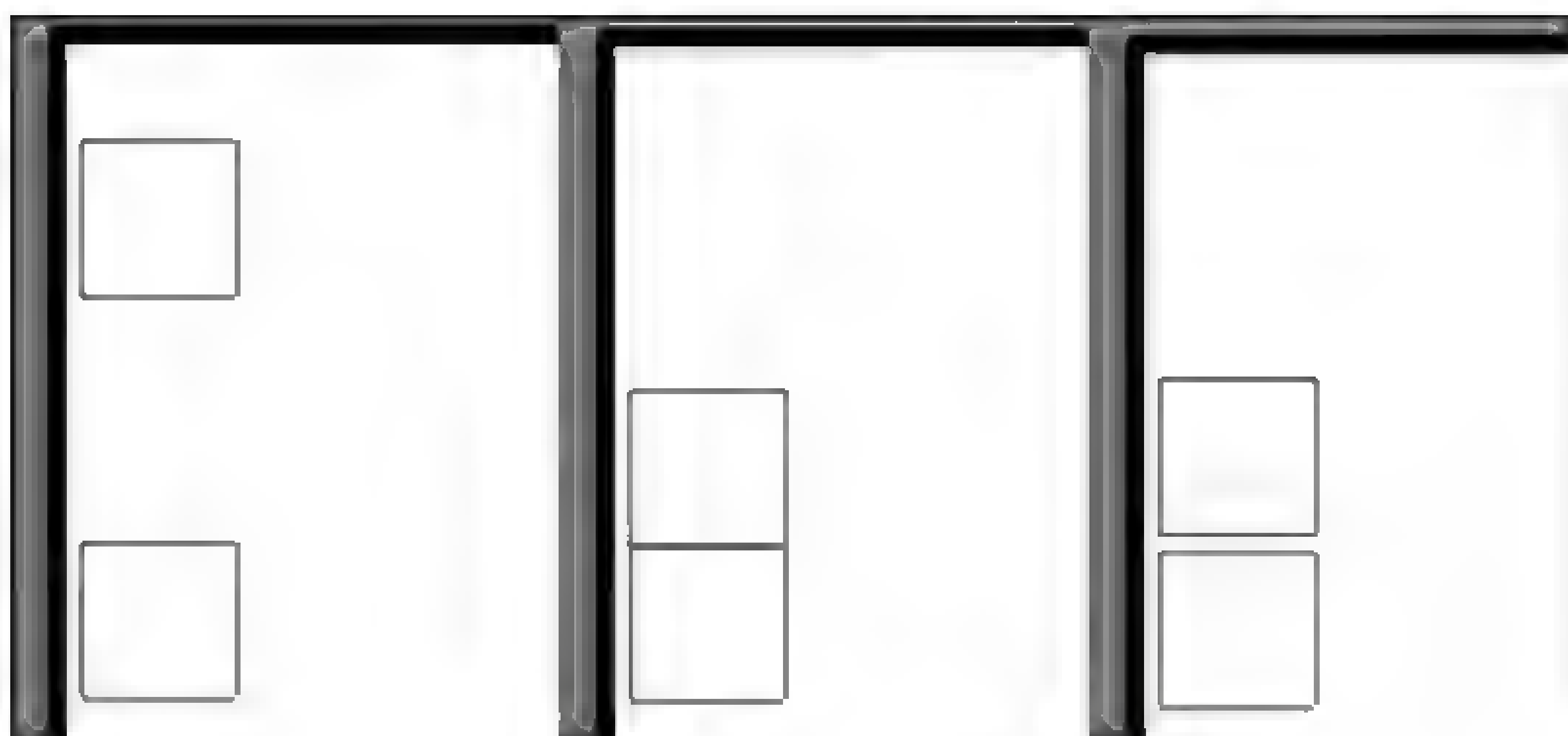


图 7-7　矩形 Body 项目运行效果图

7.6 让物体在屏幕中展现

从 7.4 节“创建 Box2D 物理世界”到 7.5 节“创建矩形物体”这两小节中可以看出，貌似这一切都与屏幕绘制没有任何的关系。

是的，屏幕绘制的图形与 Box2D 无关！Box2D 引擎只负责提供物理世界的模拟数据。换言之，如果想让屏幕中显示一个附有重力的图形，那么则需要一个重力物体运动轨迹的数据，而 Box2D 提供的 Body 正是一个拥有重力的物体。通过将 Body 在模拟的物理世界中的运动数据传给绘制的图形，绘制的图形就会沿着提供的运动轨迹来运行，也就相当于图形拥有了重力。

```
Vec2 position = body.getPosition();
polygonX = position.x * RATE-polygonWidth/2;
polygonY = position.y * RATE-polygonHeight/2;
```

通过 Body 的 getPosition 函数得到 Body 的中心点的位置，然后通过比例转换成像素，再分别赋值给屏幕绘制的图形 X,Y 坐标。

还要注意一点：此方法获取的是物体的中心点坐标，所以还需要将其 X 坐标减去物体的宽的一半，Y 坐标减去物体的高的一半，得到其左上角坐标。当然如果图形是以中心点进行绘制的话，就可以获取中心点直接将坐标传递给绘制的图形即可。

因为物理世界是在不断的模拟，所以也要不断去获取物体在物理世界的最新坐标，然后传递给绘制的图形，图形就会按照物体在物理世界中的运动轨迹去“运动”。

7.7 创建自定义多边形物体

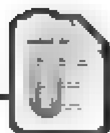
在 7.5 节中讲解了如何创建一个矩形，其实创建一个自定义多边形的过程是类似的，只是不再利用 PolygonDef.setAsBox()函数快捷创建成一个矩形形状，而是通过设置自定义的多变形的各个顶点，从而创建一个多边形。

需要注意的一点是，在 Box2D 中只能创建凸多边形，而不允许创建凹多边形!

下面就来看一个创建自定义三角形的范例，项目对应的源代码为“7-7（添加自定义多边形）”。

首先，添加一个 createMyShape 函数，代码如下：

```
public Body createMyShape(float[] vertices, float x, float y, float w,
        float h, boolean isStatic) {
        // ---创建三角形皮肤
        PolygonDef cd = new PolygonDef(); // 实例一个三角形的皮肤
        if (isStatic) {
                cd.density = 0; // 设置三角形为静态
        } else {
                cd.density = 1; // 设置三角形的质量
        }
        cd.friction = 0.8f; // 设置三角形的摩擦力
        cd.restitution = 0.3f; // 设置三角形的恢复力
        //设置三角形的每个顶点
        cd.addVertex(new Vec2(vertices[0], vertices[1]));
        cd.addVertex(new Vec2(vertices[2], vertices[3]));
        cd.addVertex(new Vec2(vertices[4], vertices[5]));
        // ---创建刚体
        BodyDef bd = new BodyDef(); // 实例一个刚体对象
        // 设置刚体的坐标
        bd.position.set((x + w / 2) / RATE, (y + h / 2) / RATE);
        // ---创建 Body（物体）
        Body body = world.createBody(bd); // 物理世界创建物体
        body.createShape(cd); // 为 Body 添加皮肤
        body.setMassFromShapes(); // 将整个物体计算打包
        return body;
}
```



提示

Box2D 在将所有顶点进行闭合时，是按照顺时针进行顶点连接的。

项目运行效果如图 7-8 所示。

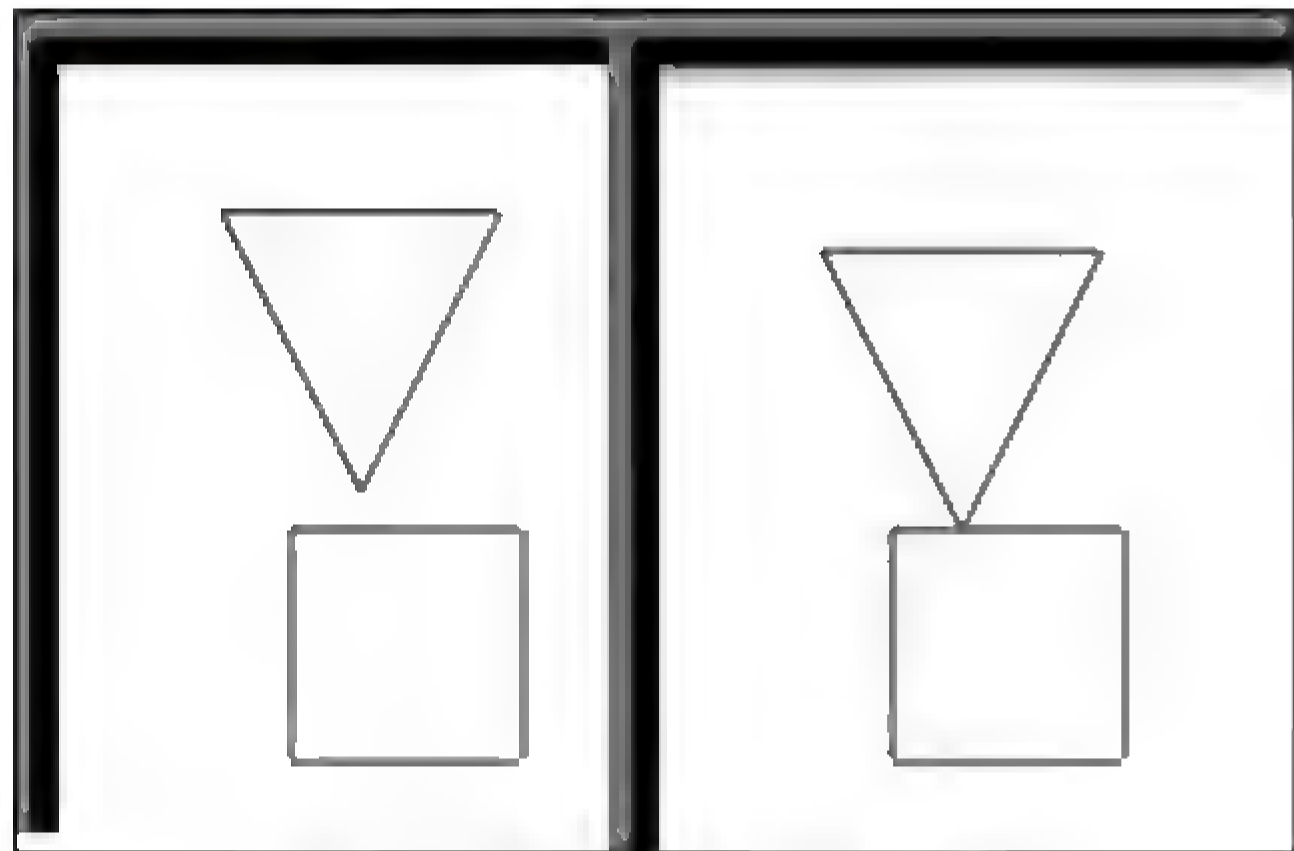


图 7-8 添加自定义多边形项目截图

大家看到图 7-8 所示的效果，可能在纳闷为什么三角形立在了矩形上。其实大家不要惊讶，虽然这种现象在现实中基本是不可能的，但是在 Box2D 这个模拟的物理世界中都是理想化的物体效果，所以这种理想化的物理状态也就见怪不怪了。

7.8 物理世界中的物体角度

前面介绍了 Body 的 X、Y 坐标属性，在现实生活两个物体发生碰撞后，除了位置坐标发生偏移外，碰撞的两个物体角度一般也会发生改变。所以，在 Box2D 模拟的物理世界中，物体（Body）除了拥有位置 X、Y 坐标这一重要属性外，还具有角度属性。

通过 Body 类的 getAngle()方法能得到 Body 的弧度，通过弧度与角度之间的转换公式来转换成角度，然后利用其角度旋转绘制的图形角度即可：

- 弧度→角度：body.getAngle()/180*Math.PI;
- 角度→弧度：body.getAngle()/Math.PI*180。

修改 7.7 节的项目代码，添加一个静态矩形 Body 放在其自定义的三角形的下方，让其发生碰撞，观察其角度的变化，如图 7-9 所示，本节项目对应的源代码为“7-7（添加自定义多边形）”。

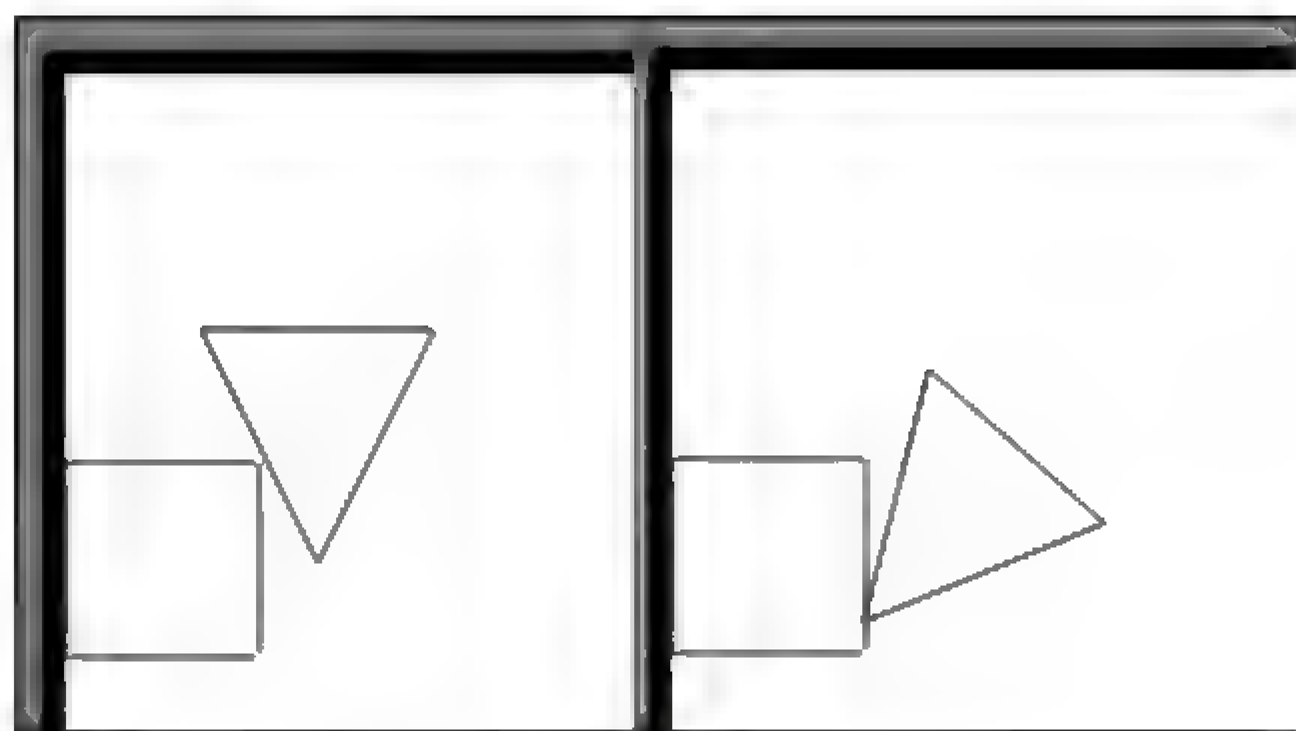


图 7-9　Body 角度的改变前后对比图

7.9 创建圆形物体

创建圆形 Body 与创建矩形也类似，只是皮肤使用 CircleDef 来创建，然后设置皮肤的半径即可，下面就来创建一个自定义三角形，对应的源代码为“7-9（在物理世界中添加圆形）”。

在项目中添加一个 createCircle 函数，代码如下：

```
public Body createCircle(float x, float y, float r, boolean isStatic) {
        // ---创建圆形皮肤
        CircleDef cd = new CircleDef(); // 实例一个圆形的皮肤
        if (isStatic) {
                cd.density = 0; // 设置圆形为静态
        } else {
                cd.density = 1; // 设置圆形的质量
        }
        cd.friction = 0.8f; // 设置圆形的摩擦力
        cd.restitution = 0.3f; // 设置圆形的恢复力
        cd.radius = r / RATE; // 设置圆形的半径
        // ---创建刚体
        BodyDef bd = new BodyDef(); // 实例一个刚体对象
        bd.position.set((x + r) / RATE, (y + r) / RATE); // 设置刚体的坐标
        // ---创建 Body（物体）
        Body body = world.createBody(bd); // 物理世界创建物体
        body.createShape(cd); // 为 Body 添加皮肤
        body.setMassFromShapes(); // 将整个物体计算打包
        return body;
}
```

在 Android 中 Canvas 常用的有两种绘制圆形的方式，一种利用 drawArc，另外一种是利

用 drawCircle，这两种虽然一个是绘制椭圆，一个是绘制圆形，但是两种都可以绘制圆形，其主要的区别在于 drawCircle 函数是要求传入圆心的坐标。

所以，在使用 body.getPosition()将 Body 中心点坐标传递给绘制的图形坐标时，其 Body 的 X,Y 坐标是否要进行减去半径，完全取决于绘制图形的方式。如果绘制圆形是使用 drawCircle 进行绘制，那么获取的 Body 的中心点坐标可以直接传递给绘制的圆形坐标；如果使用了 drawArc 函数，则需要在 Body 与绘制图形的坐标传递值时，X、Y 都需要减去半径长。

项目运行效果如图 7-10 所示。

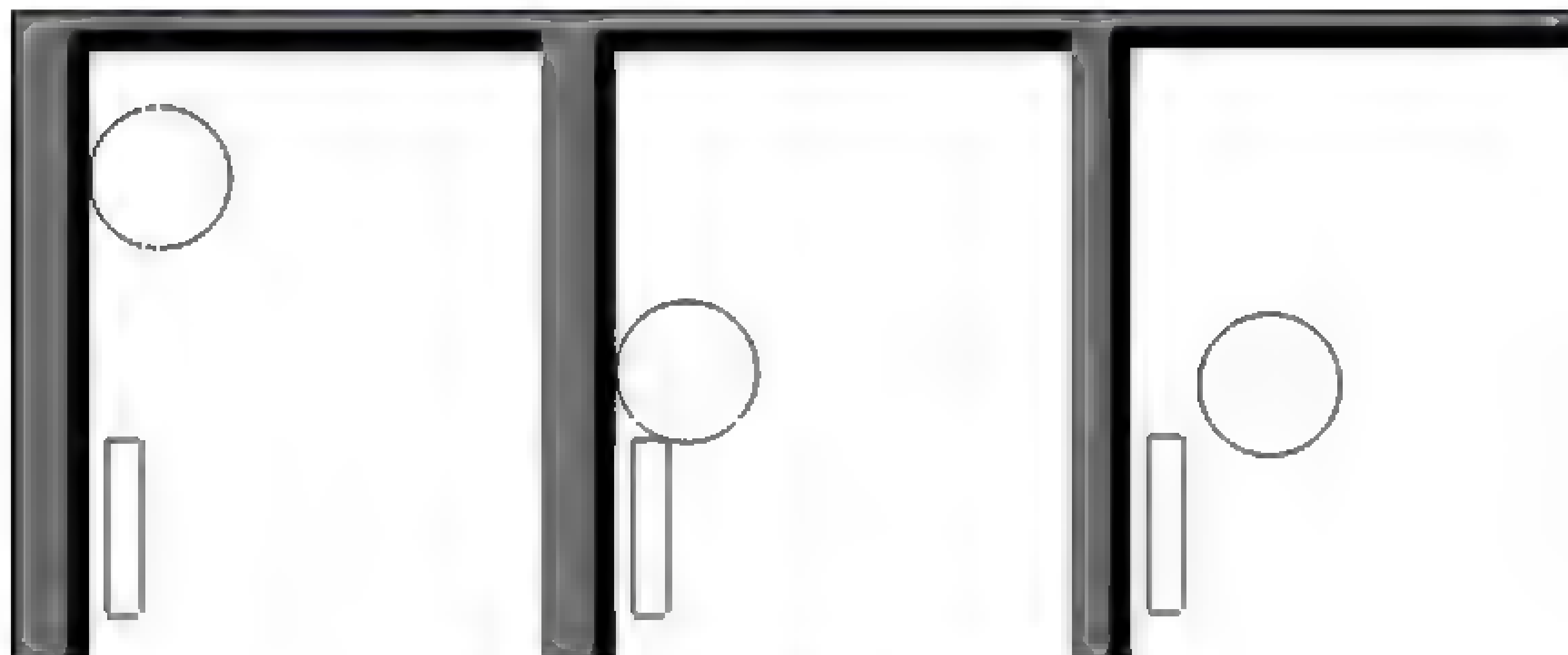


图 7-10 添加圆形 Body 项目运行效果图

7.10 多个 Body 的数据赋值

在前面的章节中，不管是创建矩形 Body，还是圆形 Body，为了让其 Body 的坐标、角度等属性能传递给绘制的图形，都要声明其对应的一个 Body 实例。但是如果 Body 需要创建很多个的时候，以前的数据传递方式就显的很笨拙，工作量也很大。

7.10.1 遍历 Body

在 World 类中有两个常用的函数：

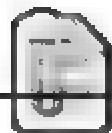
- world.getBodyCount()：此函数返回 world 中所有 body 的数量，但是要注意，新建一个物理世界，使用此方法时，不会得到零，而是返回一个 1；
- world.getBodyList()：此函数表面上看应该是返回物理世界中的 Body 的链表，其实它返回的是一个 Body 对象，就是 World 中 Body 链表的表头。

既然 world 中的 Body 是以链表方式存放的，那就应该能获取到下一个 Body 的函数。没

错！在 Body 类中有个 body.m_next 函数，此函数就是用来索引 Body 链表的下一个 Body。使用这个函数，就可以通过循环，遍历出 world 中所有的 Body。

下面就来实现多个 Body 遍历绘制，项目对应的源代码为“7-10-1（遍历 Body）”。

```
//得到 Body 链表的表头
Body body = world.getBodyList();
//通过 world.getBodyCount()得到循环遍历 Body 的次数
for (int i = 1; i < world.getBodyCount(); i++) {
        //得到当前 body 的角度
        float angle = (float) (body.getAngle() * 180 / Math.PI);
        //得到当前 body 的质点 X 坐标
        float bodyCenterX = body.getPosition().x*RATE;
        //得到当前 body 的质点 Y 坐标
        float bodyCenterY = body.getPosition().y*RATE;
        //链表指向下一个 body
        body = body.m_next;
}
```



提示

for 循环是从 1 开始的，因为 world.getBodyCount()默认返回 1。

通过此方式，可以省去 Body 与绘制图形数据传递的步骤，直接在绘制图形时，遍历取出每个 Body 的坐标等属性作为绘制图形参数值。

利用 Body 链表遍历 Body 虽然方便，省去了很多 Body 对象的声明，但是存在一个缺点：遍历获取每个 Body 的坐标与角度的逻辑代码都放在了绘制图形中，会造成其他图形的绘制延迟。

项目运行效果如图 7-11 所示。

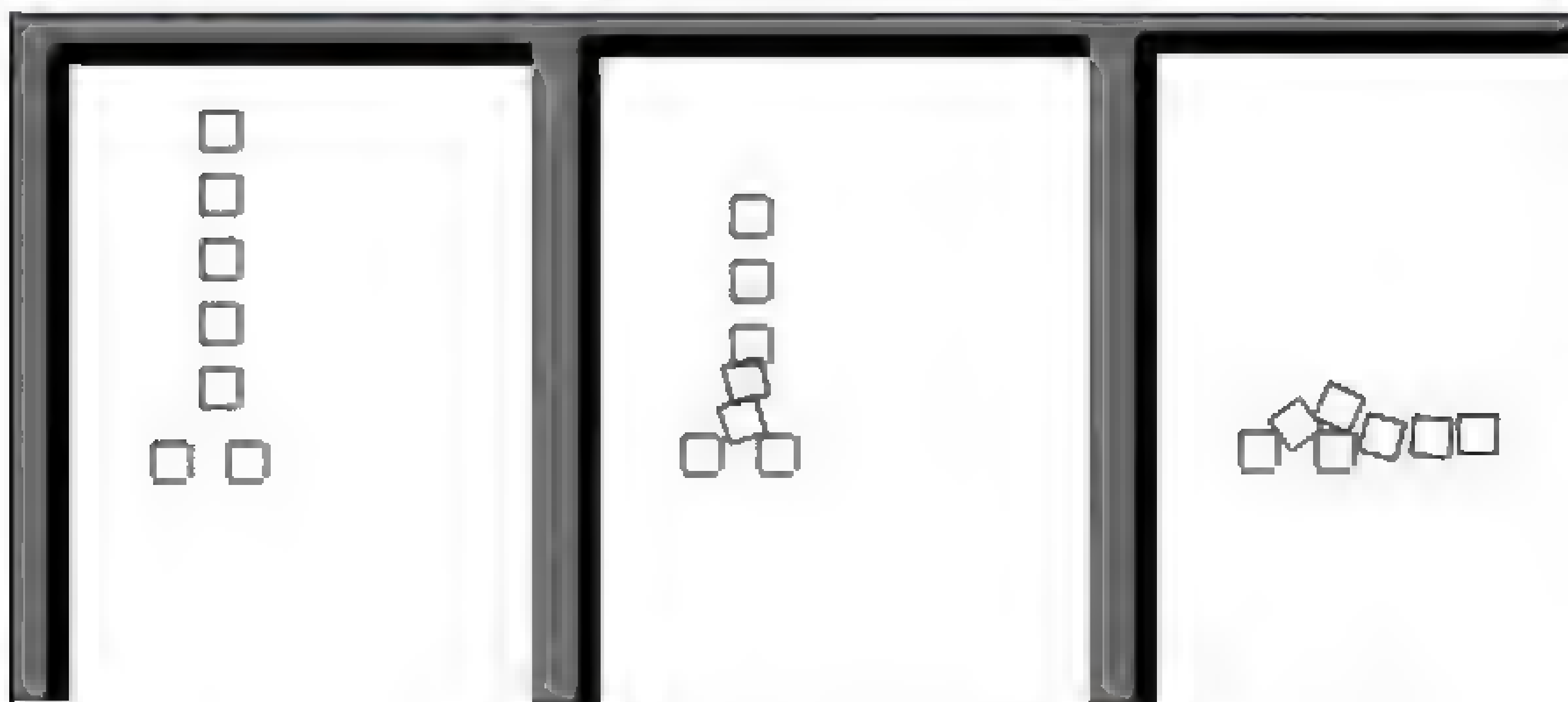


图 7-11　遍历 Body 项目效果图

7.10.2　自定义类关联 Body

在游戏开发中，都会有很多自定义的类，例如飞行射击类型的游戏中的飞机、子弹等都会是独立的类，那么下面就来实现对自定义类型进行遍历数据。

首先介绍 Body 的一个属性“m_userData”，它是一个 Object 类型，其主要用途是保存一个 Object 实例，这样就可以将自定义的类型保存到 Body 的这个 m_userData 里，通过遍历 Body 时取出其实例并进行操作。这么一说，大家就大概知道如何使用了，下面通过遍历自定义图片类来进行更加详细的讲解，项目对应的源代码为“7-10-2（Body 的 m_userData）”。

新建一个图片类“BitmapBody.java”：

```
public class BitmapBody {
      private Bitmap bmp;// 图片
      private float x, y, angle;// 图片的坐标和角度

      public BitmapBody(Bitmap bmp, float x, float y) {
             this.bmp = bmp;
             this.x = x;
             this.y = y;
      }
      // 绘制图片
      public void draw(Canvas canvas, Paint paint) {
             canvas.save();
canvas.rotate(angle, x + bmp.getWidth() / 2,
y + bmp.getHeight() / 2);
             canvas.drawBitmap(bmp, x, y, paint);
             canvas.restore();
      }
      // 设置角度
      public void setAngle(float angele) {
             this.angle = angele;
      }
      // 设置 X 轴坐标
      public void setX(float bodyX) {
             this.x = bodyX;
      }
      // 设置 Y 轴坐标
      public void setY(float y) {
             this.y = y;
      }
      // 获取图片的宽
      public int getW() {
             return bmp.getWidth();
      }
      // 获取图片的高
```

```
	public int getH() {
		return bmp.getHeight();
	}
}
```

此类很简单，就不多做说明。下面来看如何将自定义的图片类关联其 Body。添加一个 createCircle 函数，代码如下：

```
public Body                 (Bitmap bmp, float x, float y, float width,
		float height, boolean isStatic) {
	// ---创建图片 Body 皮肤
	PolygonDef pd = new PolygonDef(); // 实例一个图片 Body 的皮肤
	if (isStatic) {
		pd.density = 0; // 设置图片 Body 为静态
	} else {
		pd.density = 1; // 设置图片 Body 的质量
	}
	pd.friction = 0.8f; // 设置图片 Body 的摩擦力
	pd.restitution = 0.3f; // 设置图片 Body 的恢复力
	// 设置图片 Body 快捷成盒子(矩形)
	// 两个参数为图片 Body 宽高的一半
	pd.setAsBox(width / 2 / RATE, height / 2 / RATE);
	// ---创建刚体
	BodyDef bd = new BodyDef(); // 实例一个刚体对象
	// 设置刚体的坐标
	bd.position.set((x + width / 2) / RATE, (y + height / 2) / RATE);
	// ---创建 Body（物体）
	Body body = world.createBody(bd); // 物理世界创建物体
	//在 body 中保存自定义类
	body.m_userData = new BitmapBody(bmp, x, y);
	body.createShape(pd); // 为 Body 添加皮肤
	body.setMassFromShapes(); // 将整个物体计算打包
	return body;
}
```

创建 Body 的过程与创建一个矩形 Body 基本一致，唯一不同点就是将自定义类赋值给了 Body 的“m_userData”属性。

下面来看遍历 Body 与自定义图片类数据的传递：

```
// 得到 Body 链表的表头
Body body = world.getBodyList();
// 通过 world.getBodyCount()得到循环遍历 Body 的次数
for (int i = 1; i < world.getBodyCount(); i++) {
	// 从 body 中获取其自定义的 BitmapBody 实例
	BitmapBody bb = (BitmapBody) body.m_userData;
	// 得到当前 body 的角度
```

```
        float angele = (float) (body.getAngle() * 180 / Math.PI);
        // 得到当前 body 的质点 X 坐标
        float bodyX = body.getPosition().x * RATE - bb.getW() / 2;
        // 得到当前 body 的质点 Y 坐标
        float bodyY = body.getPosition().y * RATE - bb.getH() / 2;
        // 链表指向下一个 body
        bb.setAngle(angele);
        bb.setX(bodyX);
        bb.setY(bodyY);
        // 链表指向下一个 body
        body = body.m_next;
}
```

然后看绘图方式：

```
// 得到 Body 链表的表头
Body body = world.getBodyList();
// 通过 world.getBodyCount()得到循环遍历 Body 的次数
for (int i = 1; i < world.getBodyCount(); i++) {
        // 从 body 中获取其自定义的 BitmapBody 实例
        BitmapBody bb = (BitmapBody) body.m_userData;
        // 调用自定义图片类的 draw 方法
        bb.draw(canvas, paint);
        // 链表指向下一个 body
        body = body.m_next;
}
```

最后来看项目运行的效果，如图 7-12 所示。

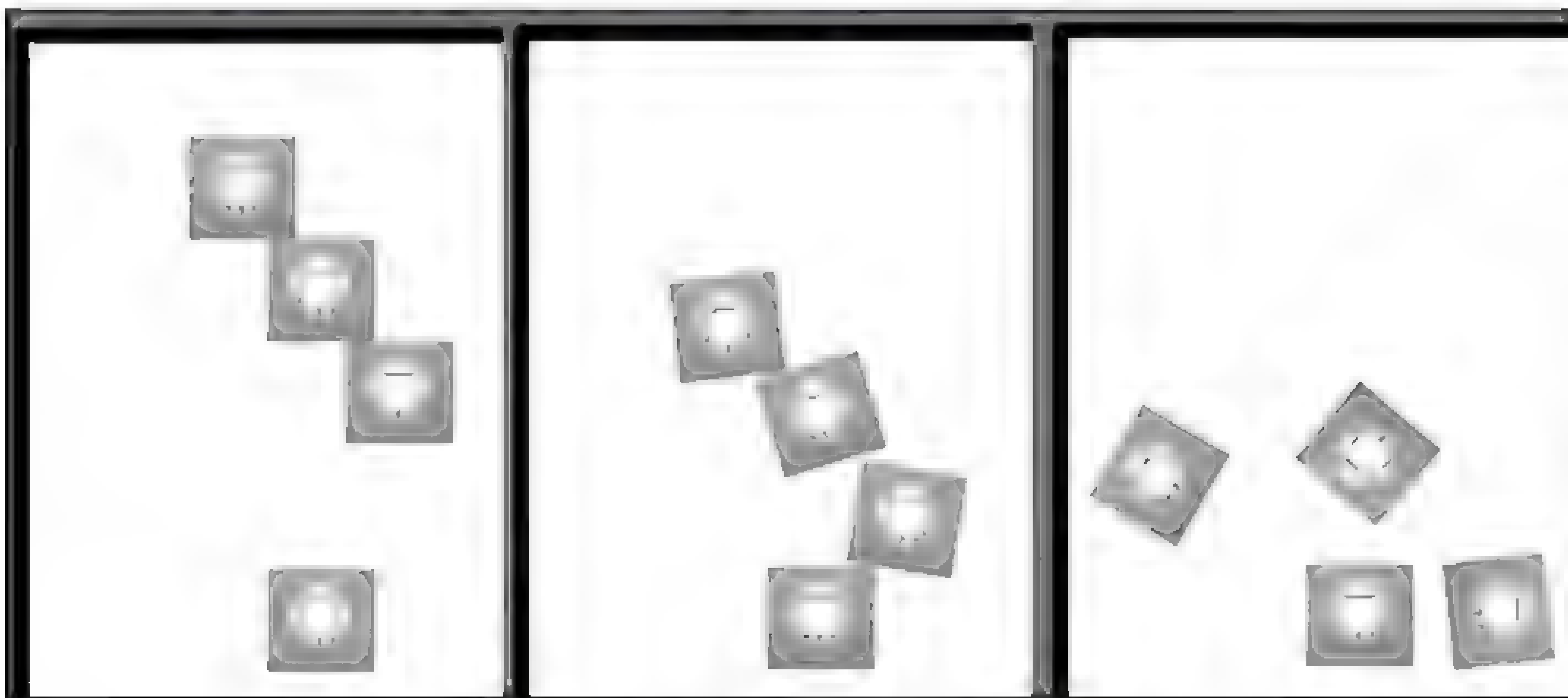


图 7-12　自定义图片类项目运行效果图

7.11 设置 Body 坐标与给 Body 施加力

7.11.1 手动设置 Body 的坐标

如需要设定 Body 的坐标，使用 Body 类中下面这个函数：

```
setXForm(Vec2 position, float angle);
```

这个函数的两个参数含义如下：

- 第一个参数：指的是 Body 在物理世界中的坐标；
- 第二个参数：指的是 Body 的角度。

7.11.2 给 Body 施加力

在前面的章节中学习了在物理世界中如何添加物体 Body 的方法。那么，如果想让物体 Body 单方向地沿着 X 轴或者 Y 轴移动，或者让 Body 沿着抛物线运动等等，该怎么做呢？

在 Box2D 的 Body 类中，提供了一个给 Body 施加力的函数：

```
applyForce(Vec2 force, Vec2 point);
```

此方法需要传入两个 Vec2 向量实例作为参数：

- 第一个参数：表示力的 X 与 Y 轴的力度，其值拥有“+”“–”属性；
- 第二个参数：表示当前调用此方法的 Body 所在物理世界的位置。

而使用 getWorldCenter()方法可以获取 Body 在物理世界中所处的位置。看到这个方法可以想到，如果需要使 Body 按照一个抛物线进行运动，那么只要施加一个 X 轴方向的力度，一个 Y 轴方向的力度，然后通过调整力度值的大小和力度值的方向（正负号）就可以组合出很多的运动轨迹。

下面就来做一个简单的“炮轰小球”的实例，项目对应的源代码为“7-11（为 Body 施加力）”。

先看项目运行效果，如图 7-13 所示。

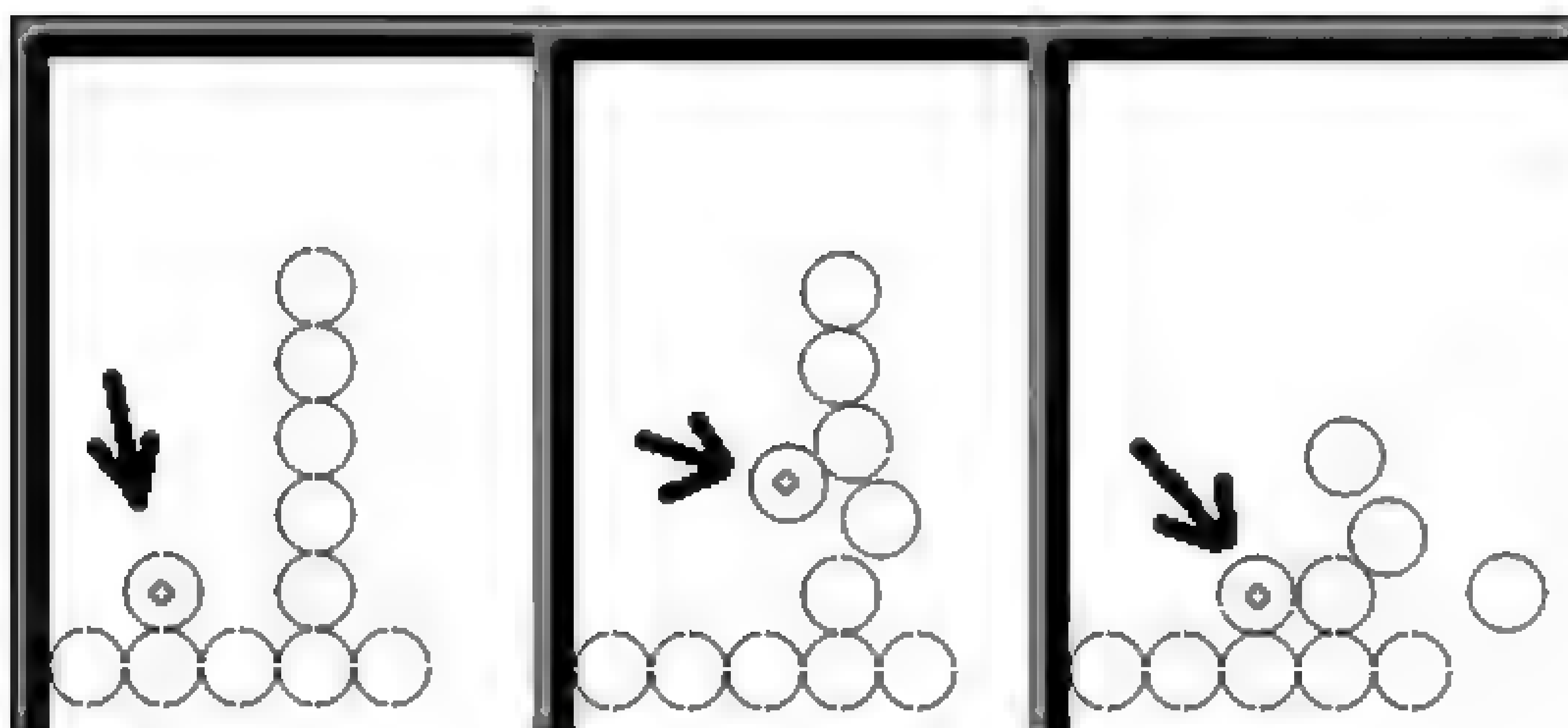


图 7-13　给 Body 施加力

具体实现步骤如下：

步骤1　首先创建“MyCircle”圆形类，便于 Body 遍历传值，代码如下所示。

```
public class MyCircle {
	//圆形的宽高与半径
	float x, y, r;
	public MyCircle(float x, float y, float r) {
		this.x = x;
		this.y = y;
		this.r = r;
	}
	//设置圆形的 X 坐标
	public void setX(float x) {
		this.x = x;
	}
	//设置半径的 Y 坐标
	public void setY(float y) {
		this.y = y;
	}
	//绘制圆形
	public void draw(Canvas canvas, Paint paint) {
		canvas.drawArc(new RectF(x, y, x + 2*r, y + 2*r),
				0, 360, true, paint);
	}
}
```

圆形类设计的比较简单，值得注意的是绘制函数 draw。这里以 drawArc 的方式去绘制圆形，所以当 Body 与图形传 X、Y 坐标时，要记得分别减去 Body 的宽和高的一半。如果这里使用 drawCircle 的话，那么 Body 与图形传坐标值就不需要做减值操作了，因为默认获取的是 Body 的中心点坐标。

步骤2　创建圆形 Body，代码如下所示。

```
// 实例需要施加力的body小球，因为在后续的按键处理中，
//我们需对齐小球施加力的操作：所以body1声明为成员变量；
body1 = createCircle(20, 120, 10, false);
// ----在物理世界中添加多个动态圆形Body
for (int i = 0; i < 5; i++) {
      createCircle(60, 50+i * 17, 10, false);
}
// 添加屏幕下方添加多个静态物体
for (int i = 0; i < 5; i++) {
      createCircle(i * 20, 150, 10, true);
}
// 创建圆形Body函数
public Body createCircle(float x, float y, float r, boolean isStatic){
    CircleDef cd = new CircleDef();
    if (isStatic) {
        cd.density = 0;
    } else {
      cd.density = 1;
    }
    cd.friction = 0.8f;
    cd.restitution = 0.3f;
    cd.radius = r / RATE;
    BodyDef bd = new BodyDef();
    bd.position.set((x + r) / RATE, (y + r) / RATE);
    Body body = world.createBody(bd);
    body.m_userData = new MyCircle(x, y, r);
    body.createShape(cd);
    body.setMassFromShapes();
    return body;
}
```

步骤3 圆形Body与圆形图形坐标传值遍历，代码如下所示。

```
// 取出body链表表头
Body body = world.getBodyList();
for (int i = 1; i < world.getBodyCount(); i++) {
      // 设置MyCircle的X，Y坐标
      MyCircle mc = (MyCircle) body.m_userData;
      mc.setX(body.getPosition().x * RATE - mc.r);
      mc.setY(body.getPosition().y * RATE - mc.r);
      // 将链表指针指向下一个body元素
      body = body.m_next;
}
```

步骤4 绘制Body，代码如下所示。

```
//遍历取出body，通过body的m_userData属性得到其MyCircle实例
```

```
Body body = world.getBodyList();
for (int i = 1; i < world.getBodyCount(); i++) {
      //每个MyCircle实例调用其绘制函数
      ((MyCircle) body.m_userData).draw(canvas, paint);
      body = body.m_next;
}
```

步骤5 按键事件中为小球施加力，代码如下所示。

```
@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
      Vec2 vForce = new Vec2(150, -150);
      body1.applyForce(vForce, body1.getWorldCenter());
      return true;
}
```

当手动为一个 Body 施加一个力，即使它在受力之前处于静止正休眠状态，也会被唤醒，但是要注意最好不要给静态物体 Body 施加力，因为对一个静态物体施加力，虽然这个力存在，但其实是起不到作用的，静态物体始终静止。

7.12 Body碰撞监听、筛选与Body传感器

7.12.1 Body碰撞接触点监听

在 Box2D 中提供了一个碰撞监听器接口“ContactListener”，使用接口需要重写它的四个抽象函数：

```
@Override
public void add(ContactPoint arg0) {
}
@Override
public void persist(ContactPoint arg0) {
}
@Override
public void remove(ContactPoint arg0) {
}
@Override
public void result(ContactResult arg0) {
}
```

ContactListener 接口的 4 个函数的含义为：

- add 函数：发生碰撞，有新的接触点时响应的函数；
- persist 函数：当已存在的接触点仍存在时响应的函数；
- remove 函数：当存在的接触点被删除时响应的函数；
- result 函数：每次时间步监听，如仍有触点存在则被响应；

通过这 4 个函数中的参数 ContactPoint 类的实例，可以获取到相互之间碰撞的两个 Body 实例等等，获取 Body 的方法如下所示：

```
Body body1 =arg0.shape1.getBody();
Body body2 =arg0.shape2.getBody();
```

最后让物理世界 World 实例去绑定监听器即可：

```
world.setContactListener(ContactListener listener);
```

7.12.2 Body 碰撞筛选

通过上一小节讲解的方法，就可以完成对物理世界的碰撞监听；但是有些情况下，我们并不想让每个 Body 之间都发生物理碰撞，或者说想指定 Body 之间是否能发生碰撞的关系，这时我们需要来熟悉 Box2D 中的 FilterData 类。

1. FilterDate 类

FilterData 类一般不直接使用，只需要设置此类中的属性，此类的源码如下：

```
public class FilterData {
      public int categoryBits;
      public int maskBits;
      public int groupIndex;
      public void set(FilterData fd) {
             categoryBits = fd.categoryBits;
             maskBits = fd.maskBits;
             groupIndex = fd.groupIndex;
      }
}
```

FilterData 是 Box2D 提供的一个碰撞接触的数据过滤类，类的结构很简单，但是其中有的 3 个属性格外的重要，下面对 FilterData 类的 3 个属性进行说明。

（1）groupIndex 属性

groupIndex 属性文如其名，是个分组下标。所谓分组，其实就是 Box2D 检测 Body 之间是否发生碰撞的一个判定条件。

例如，物理世界中有两个 Body，分别为 body1、body2。假设对两个 Body 设置其属性值：

```
body1 的 groupIndex=1, body2 的 groupIndex=1;
```

那么这两个 Body 之间会发生碰撞。

但是如果这么设置:

```
body1 的 groupIndex=-1, body2 的 groupIndex=-1;
```

那么这两个 Body 之间则永远不会发生碰撞。

这里的 groupIndex 属性的作用可以用 3 点来概括:

- 多个 Body 的 groupIndex 属性值为正值(>=0), Body 之间永远能发生碰撞;
- 多个 Body 的 groupIndex 属性值为负值(<0), 同一组(groupIndex 值相同)的 Body 之间永远不会发生碰撞; 不同组(groupIndex 值不相同)的 Body 之间会发生碰撞;
- 多个 Body 的 groupIndex 属性值各不相同时, 都会产生碰撞; 如果有相同组时, 其判定方法通过前面两点来进行判定。

使用 groupIndex 属性需要注意两点: 一是 groupIndex 的值不能大于一个字节; 二是同一个 Body 设置多个 groundIndex 的值, 默认取最后一次设定的值。

(2) categoryBits 和 maskBits 属性

groupIndex 是最优先的碰撞检测, 除了通过组进行筛选碰撞之外, 还可以通过指定种类来进行筛选碰撞:

- categoryBits: 设置碰撞种类;
- maskBits: 指定碰撞种类。

假设有 3 个 Body: body1、body2 和 body3。其中 body1 的 categoryBits=2, body2 的 categoryBits=4, 那么 body3 如果想与 body1 和 body2 都发生碰撞, 则 body3 的 maskBits 值应该为 6 (body1 和 body2 的 categoryBits 之和)。这里需要注意:

- categoryBits 属性值必须为 2 的倍数, 否则可能出现判定失物;
- 从官方提供的数据了解到, 对于 groupIndex 分组, Box2D 只能同时支持 16 个种类。

前面为大家讲述了 FilterData 类的 3 个重要属性, 下面就来讲解如何为 Body 设置这几个属性值。设置的方式有两种:

①在创建 Body 的皮肤时进行设置, 这里举例创建圆形 Body 皮肤时的设置:

```
CircleDef cd = new CircleDef();
```

- 设置 groupIndex: cd.filter.groupIndex=1;
- 设置 categoryBits: cd.filter.categoryBits=1;
- 设置 maskBits: cd.filter.maskBits=1。

②通过已创建的 Body 进行设置, 这里假设已经创建好了一个 body1:

- 设置 groupIndex:

```
body1.getShapeList().getFilterData().groupIndex = 1;
```

- 设置 categoryBits:

```
body1.getShapeList().getFilterData().categoryBits = 1;
```

- 设置 maskBits:

```
body1.getShapeList().getFilterData().maskBits = 1;
```

下面来看一个创建多个 Body 的范例，并设置其分组与种类属性，项目对应的源代码为“7-12（Body 碰撞监听）”。

```
// ----在物理世界中添加两个动态圆形 Body
Body body1 = createCircle(39, 17, 10, false);
Body body2 = createCircle(30, 47, 10, false);
// 定义 body1 分组 1
body1.getShapeList().getFilterData().groupIndex = 1;
// 指定 body1 碰撞种类为 2
body1.getShapeList().getFilterData().maskBits = 2;
// 定义 body2 分组 2
body2.getShapeList().getFilterData().groupIndex = 2;
// 定义 body2 种类为 2
body2.getShapeList().getFilterData().categoryBits = 2;
// 添加屏幕下方添加多个静态物体
for (int i = 0; i < 5; i++) {
      Body body = createCircle(i * 20, 100, 10, true);
      // 定义全部静态 body 分组为 3
      body.getShapeList().getFilterData().groupIndex = 3;
      // 定义全部静态 body 种类为 4
      body.getShapeList().getFilterData().categoryBits = 4;
}
```

在上面的代码中，省略了 Body 的创建代码。对代码进行分析，首先观察到 body1、body2 与所有的静态 body 的分组都为正值，所以它们相互之间肯定发生碰撞；然后观察到 body1 指定碰撞的种类值为 2，body2 设置了种类为 2，其他静态 body 的种类设置为 4，那么可以得到结论：

- body1 与 body2 之间能发生碰撞；
- body1 与其他静态 body 不会发生碰撞；
- body2 与静态物体会发生碰撞。

运行项目，效果如图 7-14 所示。

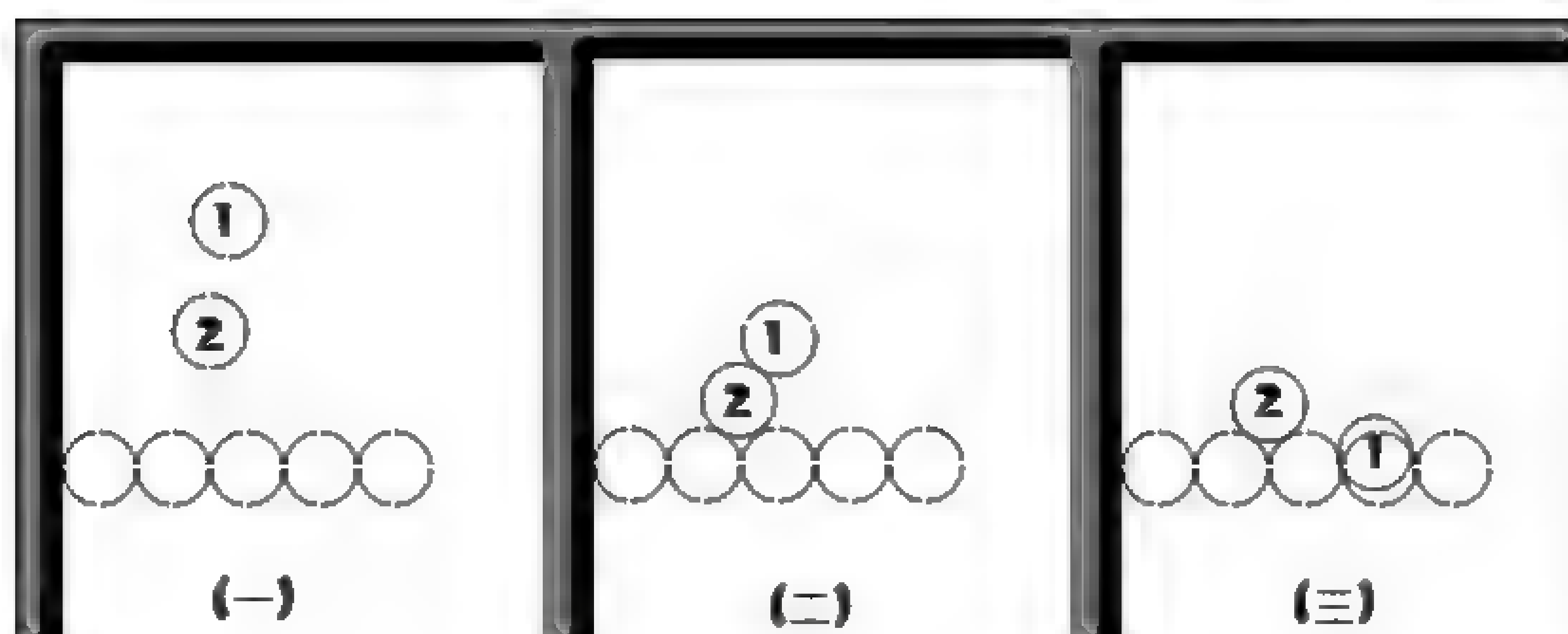


图 7-14 Body 之间的碰撞关系

2. Body 碰撞筛选监听器

在 7.12.1 小节中介绍了碰撞监听器，那么在 Box2D 提供的碰撞监听器中还有一个筛选监听器“ContactFilter”。这两个监听器的作用都是用于监听碰撞事件，而且两者的使用方式都相同，使用其接口、重写其函数、最后使用物理世界绑定其监听器。

使用筛选监听器，只需要重写一个函数：

```
@Override
public boolean shouldCollide(Shape shape1, Shape shape2) {
    return false;
}
```

默认此函数返回 false，一旦我们的物理世界绑定此筛选监听器，那么所有的 Body 之间都将失去碰撞效果。

这里需要注意：所有的 Body 之间失去碰撞效果，而不是失去碰撞检测！换言之，当两个 Body 发生碰撞时，筛选监听器仍会响应 shouldCollide 函数，但是这两个 Body 会相交，没有碰撞后可能被弹开、发生角度偏移等视觉效果。

那么我们可以在筛选监听器的 shouldCollide 函数中编写一些处理代码，例如在函数中添加如下代码（假设有物理世界中存在两个 Body，分别为 body1 与 body2，并且物理世界设置了筛选监听器）：

```
@Override
public boolean shouldCollide(Shape shape1, Shape shape2) {
    if (shape1.getBody() == body1 && shape2.getBody() == body2)
        return true;
    return false;
}
```

这么处理后，body1 与 body2 发生碰撞时就会拥有碰撞效果。碰撞筛选器比碰撞监听器更加具有扩展性，可以在碰撞筛选器中实现监听器的功能，但是由于其自由度太大，一般使用碰撞监听器就足以满足处理要求。

3. Body 传感器

Body 的传感器其实就是 Body 皮肤的一个属性，属性名为 isSensor，默认值为 false。设置其 isSensor 属性的方式有两种：

- 创建 Body 皮肤时设置 isSensor 属性;
- 利用已创建的 Body 去设置。

假设有一个 Body 实例 body1，设置方法如下：

```
body1.getShapeList().m_isSensor=true;
```

Body 传感器的作用是：一个 Body 的传感器属性如果设为真（true），此 Body 则不会与其他 Body 产生碰撞效果，但是此 Body 与其他 Body 之间的碰撞也能被监听器监听到。

例如有一款游戏中，有一个门 Body，有一个球 Body，当小球每次只要从门的一侧穿过到达另一侧时，就计数一次，当计数达到一定的次数以后则判定游戏胜利。那么针对这样一款游戏，首先想到既然需要进行次数统计，那么一定会对门与小球两个 Body 进行监听碰撞，但是又不能让两个 Body 产生碰撞效果，因为游戏中需要小球穿过门！这时就可以利用 Body 传感器属性来实现相应的代码。

其实大家也不难想到，类似上述这种情况，即使不使用 Body 的传感器属性，单单利用 Body 的组与种类属性也能做到。

7.13 关节

在 Box2D 中除了物体 Body 外，还使用到关节 Joint。关节的主要作用是用来限制和约束 Body 之间的位置、距离、速度、运动轨迹等，本节将详细讲解在 Box2D 中的 6 种关节作用和其实现方法。

7.13.1 距离关节

距离关节（DistanceJoint）用来限制两个 Body 的质心距离永久保持不变。

我们可以在项目中创建一个距离关节，项目对应的源代码为“7-13-1（距离关节）”。在项目中添加一个 createDistanceJoint 函数：

```
public DistanceJoint createDistanceJoint() {
      //创建一个距离关节数据实例
      DistanceJointDef dje = new DistanceJointDef();
      // 初始化关节数据
      dje.initialize(body1, body2, body1.getWorldCenter(),
```

```
body2.getWorldCenter());
        // dje.collideConnected=true;
        //利用世界通过传入的距离关节数据创建一个关节
        DistanceJoint dj = (DistanceJoint) world.createJoint(dje);
        return dj;
}
```

上面代码使用了 Initialize（Body body1,Body body2,Vec2 anchor1,Vec2 anchor2）方法，这个方法的前两个参数是绑定在距离关节上的两个 Body 的实例，后面两个参数是两个 Body 在物理世界的坐标。

关节的创建也与 Body 的创建相似，不能直接 new，只能通过物理世界 world 来“生产”出来。创建一个关节，都是利用 World 的 createJoint（JointDef def）函数来创建，这个函数的参数要求传入关节的数据信息。World 也是根据传入的这个关节数据信息来决定应该创建出什么类型的关节。

在关节数据的实例中一般都会用到 collideConnected 属性，此属性表示绑定在关节上的两个 Body 之间是否可以发生碰撞。

距离关节的作用一开始提到过，它可以保持两个 Body 的质心距离不变，那么这个距离就是在距离关节数据进行初始化的时候根据两个 Body 在物理世界的坐标而确定的。

运行项目，效果如图 7-15 所示。

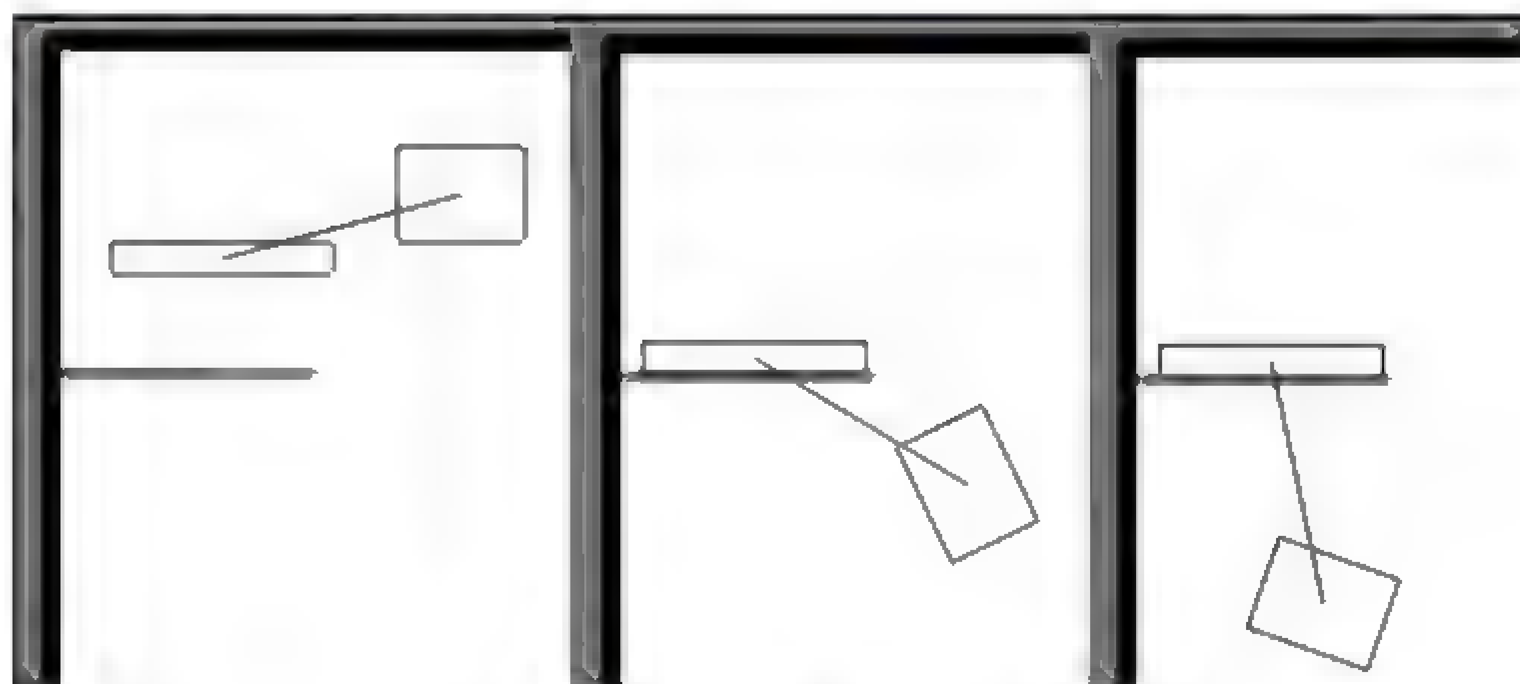


图 7-15 距离关节

关节 Joint 的常用方法和属性如下所示。

- m_body1：设置关节的 body1 实例；
- m_body2：设置关节的 body2 实例；
- dj.getBody1()：获取关节的 body1 实例；
- dj.getBody2()：获取关节的 body2 实例；
- getAnchor1()：获取关节的第一个锚点坐标；
- getAnchor2()：获取关节的第二个锚点坐标。

7.13.2 旋转关节

旋转关节（RevoluteJoint）指一个 Body 围绕另外一个 Body 进行旋转。下面举一个例子来说明如何创建一个旋转关节，示例项目对应的源代码为“7-13-2（旋转关节）”。

在项目中添加一个 createRevoluteJoint 函数，Body 创建的代码此处省略：

```
public RevoluteJoint createRevoluteJoint() {
        //创建一个旋转关节的数据实例
        RevoluteJointDef rjd = new RevoluteJointDef();
        //初始化旋转关节数据
        rjd.initialize(body1, body2, body1.getWorldCenter());
        rjd.maxMotorTorque = 1;// 马达的预期最大扭矩
        rjd.motorSpeed =20;//马达最终扭矩
        rjd.enableMotor = true;// 启动马达
        //利用 world 创建一个旋转关节
        RevoluteJoint rj = (RevoluteJoint)world.createJoint(rjd);
        return rj ;
}
```

在上面代码中，初始化旋转关节数据的函数为 Initialize（Body body1,Body body2,Vec2 anchor），它的参数含义说明如下：

- 第一个参数：做旋转运动的 Body 实例；
- 第二个参数：旋转关节中的第二个 Body 实例；
- 第三个参数：旋转锚点。

当旋转关节的数据初始化之后，旋转 Body 是不会运动的，因为没有力的作用，所以需要一个“马达”来进行驱动 Body，让 Body 开始做旋转运动。

“马达”也是旋转关节中的数据，所以利用 RevoluteJointDef 实例来进行设置，启动一个“马达”驱动旋转 Body 进行旋转运动，至少需要设置下面三个属性：

- maxMotorTorque：马达的预期最大扭矩；
- motorSpeed：马达最终扭矩；
- enableMotor：启动马达。

马达预期最大扭矩指的是 Body 在进行旋转运动开始的一个扭矩值，可以简单理解为旋转速度。马达最终扭矩指的是马达最终会以一个固定的转速来运动，而这个转速就是取决最终扭矩。当两者属性设置完毕，最后启动马达即可。

运行项目，效果如图 7-16 所示。

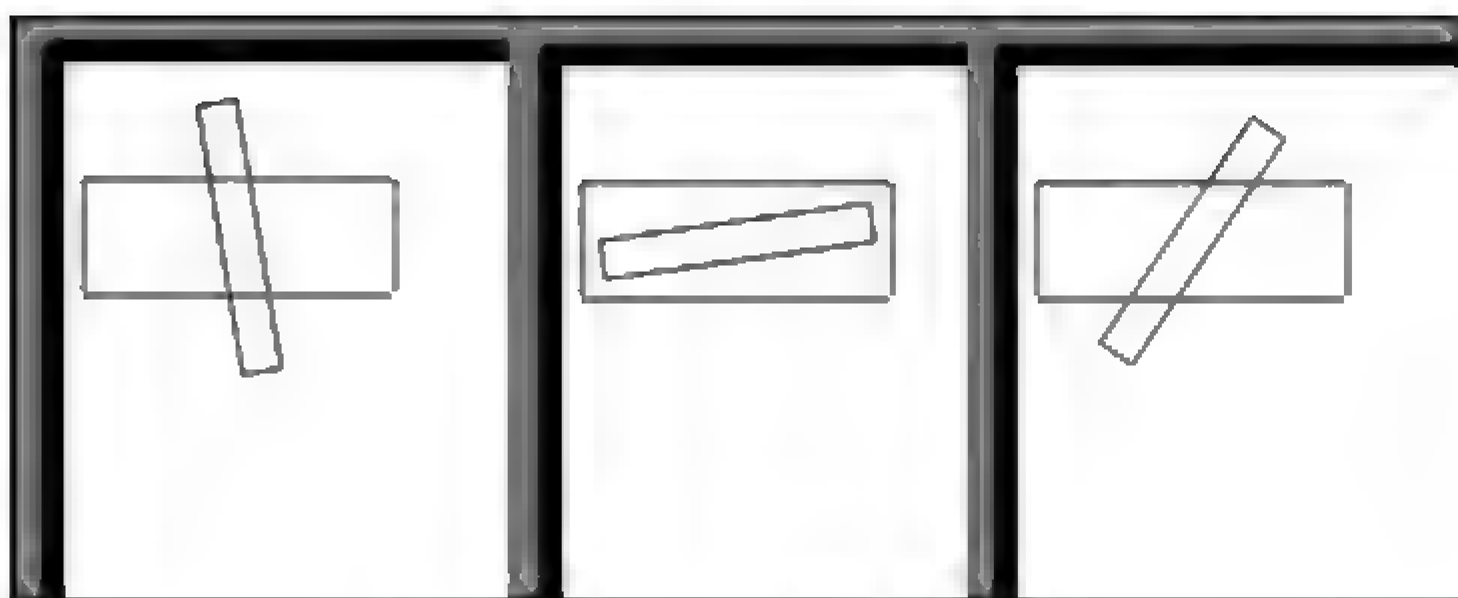


图 7-16 旋转关节

在使用旋转关节时需要注意以下几点：

- 设置的“预期扭矩” 如果比“最终扭矩”小，那么要考虑“预期扭矩”的扭矩力是否能克服重力使旋转 Body 运动，如果“预期扭矩”偏小则否则无法正常旋转；
- 马达默认转速是根据“预期最大扭矩”来决定的；
- 旋转 Body 取决于初始化旋转关节函数 Initialize 的第一个 Body 参数；
- 旋转关节中默认逆时针旋转，可通过设置扭矩来设置旋转方向；当扭矩为负数时旋转 Body 进行顺时针旋转；
- 设置预扭矩时要注意不要过小，否则当无法抗拒自身的重力的话就无法旋转。

这里，再说明一下旋转关节的角度限制问题。旋转关节是一个 Body 围绕另外一个 Body 进行旋转运动，那么除了设置预期扭矩与最终扭矩之外，还能限制旋转关节的旋转角。可以通过 RevoluteJointDef 实例来设置旋转关节的角度限制，其方法如下：

- enableLimit：是否启动关节限制；
- lowerAngle：旋转关节角的最小角度；
- upperAngle：旋转关节角的最大角度。

在 Box2D 中逆时针旋转时，旋转关节角为正；顺时针旋转时，旋转关节角为负。

7.13.3 齿轮关节

齿轮关节（GearJoint）指让两个 Body 进行齿轮咬合运动。上一小节讲解了如何创建旋转关节，本小节紧接着来讲述齿轮关节是有原因的，因为齿轮关节是需要两个旋转关节来创建完成的，所以创建齿轮关节需要分两个步骤完成，以项目源代码“7-13-3（齿轮关节）”为例说明如何创建齿轮关节。

（1）创建两个旋转关节

创建两个旋转关节的代码如下：

```
//创建第一个旋转关节
public RevoluteJoint createRevoluteJoint1() {
```

```
        RevoluteJointDef rjd = new RevoluteJointDef();
        rjd.initialize(world.getGroundBody(), body1,
body1.getWorldCenter());
        rjd.maxMotorTorque = 20;
        rjd.motorSpeed = 20;
        rjd.enableMotor = true;
        RevoluteJoint rj = (RevoluteJoint) world.createJoint(rjd);
        return rj;
}
//创建第二个旋转关节
public RevoluteJoint createRevoluteJoint2() {
        RevoluteJointDef rj = new RevoluteJointDef();
        rj.initialize(world.getGroundBody(), body2,
body2.getWorldCenter());
        return (RevoluteJoint) world.createJoint(rj);
}
```

这里创建两个旋转关节分别使用了两个方法去创建，原因在于：

在创建齿轮关节时，需要设置两个旋转关节，那么由于在齿轮关节中默认由第一个旋转关节带动第二个旋转关节，而第二个齿轮关节不需要任何设置，只需要初始化。也就是说齿轮关节中一个是“自运动的旋转关节”，另外一个是“被带动的旋转关节”，所以两个旋转关节的创建是不太一样的。

创建绑定在齿轮关节上的旋转关节需要注意以下两点：

- 齿轮关节绑定的每个旋转关节在创建时，旋转关节数据初始化时的第一个参数 body1 必须为“接地 Body”；所谓“接地 Body”，其实是 World 物理世界中自带的一个 Body，可以通过 world.getGroundBody()函数来得到。
- 在遍历 Body 一节中提到，通过 world.getBodyCount()获取的值默认返回是 1，那么，由此我们可以猜测可能在创建物理世界时，就初始化了一个接地 Body。所以即使没有往 World 里添加 Body，通过 world.getBodyCount()函数总不会返回 0。

因为齿轮关节默认从慢速开始运动，所以即使创建旋转关节时设置它的“预期最大扭矩”小于“最终扭矩值”变得无效果；但是“预期最大扭矩”与“最终扭矩值”为旋转关节转动的必须条件，虽然在齿轮关节中没有效果但是必须有其属性。

（2）创建齿轮关节

创建齿轮关节的代码如下：

```
public GearJoint createGearJoint() {
        //创建齿轮关节数据实例
        GearJointDef gjd = new GearJointDef();
        //设置齿轮关节的两个Body
        gjd.body1 = body1;
        gjd.body2 = body2;
```

```
        //设置齿轮关节绑定的两个旋转关节
        gjd.joint1 = rj1;
        gjd.joint2 = rj2;
        //设置旋转角度比
        gjd.ratio = 10;
        //通过world创建一个齿轮关节
        GearJoint gj = (GearJoint) world.createJoint(gjd);
        return gj;
}
```

齿轮关节没有初始化关节数据的方法，而是通过手动去设置 Body 与旋转关节完成。代码中的"ratio=10"表示"自运动的旋转关节"转动十周时，"被带动的旋转关节"正好旋转一周。

运行项目，效果如图 7-17 所示。

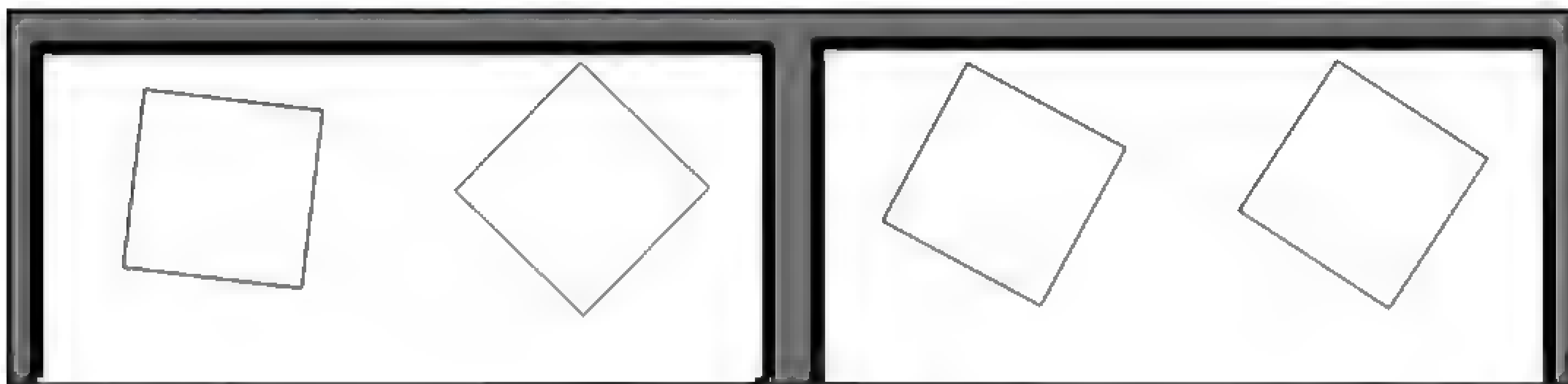


图 7-17 齿轮关节

7.13.4 滑轮关节

滑轮关节（PulleyJoint）指两个 Body 绑定滑轮关节，让两个 Body 都沿着一个世界锚点进行滑轮运动。

我们用一个范例来说明如何创建一个滑轮关节，范例项目对应的源代码为"7-13-3（滑轮关节）"。在项目中添加一个 createPulleyJointDef 函数，Body 创建的代码此处省略：

```
public PulleyJoint createPulleyJointDef() {
        //创建滑轮关节数据实例
        PulleyJointDef pjd = new PulleyJointDef();
        Vec2 ga1 = new Vec2(anchor1x/ RATE,anchor1y/ RATE);
        Vec2 ga2 = new Vec2(anchor2x/ RATE,anchor2y/ RATE);
        //初始化滑轮关节数据
        //body，两个滑轮的锚点，两个body的锚点，比例系数
        pjd.initialize(body1, body2, ga1,ga2,
                    body1.getWorldCenter(),
                    body2.getWorldCenter(), 1f);
```

```
        PulleyJoint pj = (PulleyJoint) world.createJoint(pjd);
        return pj;
}
```

createPulleyJointDef 函数的初始化方法为 Initialize（Body body1,Body body2, Vec2 ga1,Vec2 ga2,Vec2 anchor1,Vec2 anchor2,float r），其参数说明如下：

- 第一个参数：第一个 Body 实例；
- 第二个参数：第二个 Body 实例；
- 第三个参数：第一个滑轮锚点；
- 第四个参数：第二个滑轮锚点；
- 第五个参数：拉伸长度的比例。

项目运行效果如图 7-18 所示。

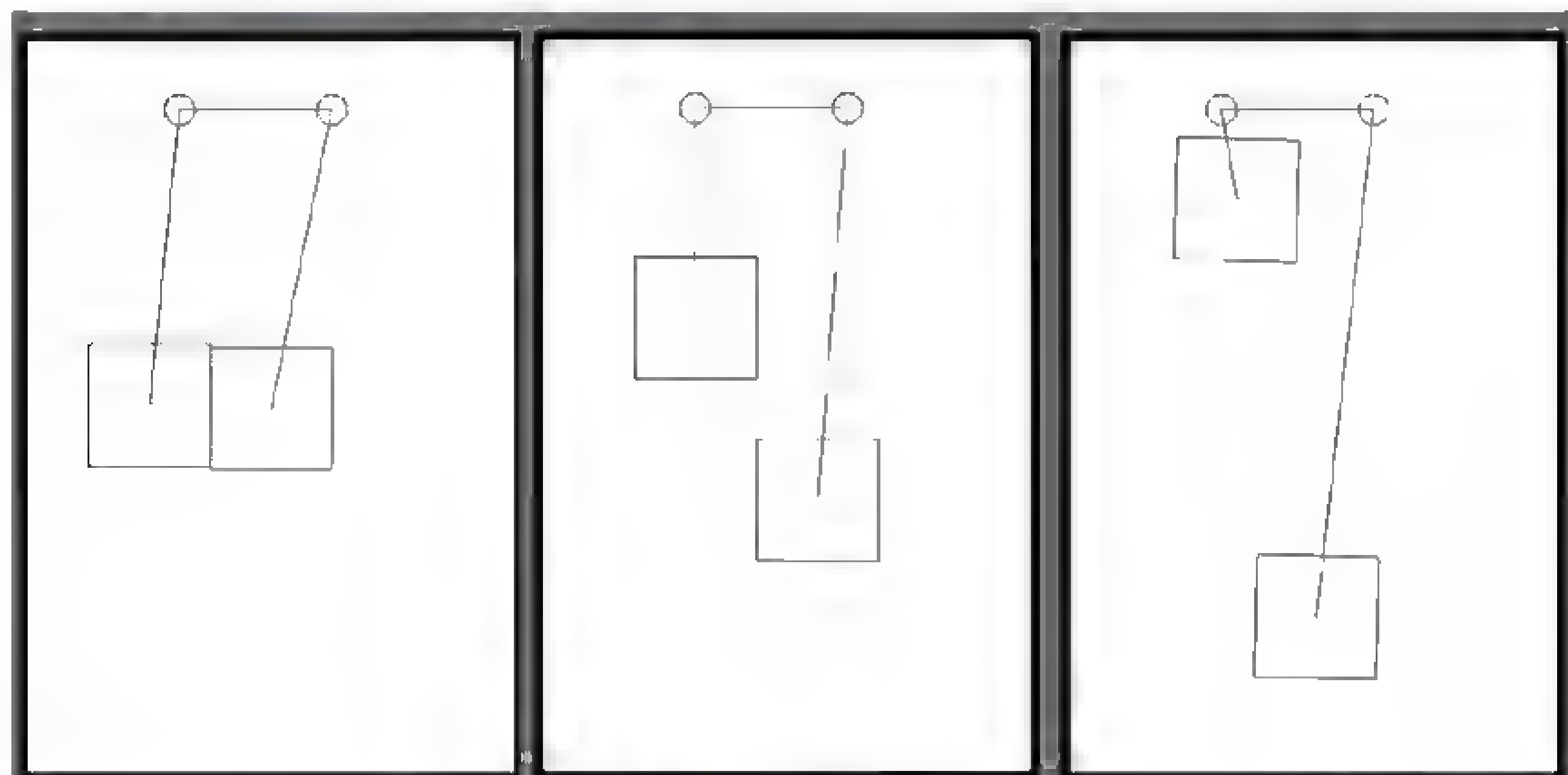


图 7-18　滑轮关节

在滑轮关节中，两个 Body 谁会向上运动，谁会向下运动，这完全取决于滑轮关节上的两个 Body 中每个 Body 的质量大小，谁重谁肯定向下运动。

现在回头解释 Initialize 函数的第五个参数拉伸长度的比例。假设当前创建了一个如图 7-19 所示的滑轮关节。

在图 7-19 中，Body2 质量大于 Body1 的质量，所以 Body1 向上运动，Body2 向下运动。

- 如果设置此滑轮关节的长度比为 1，那么当 length1=5 时，length2=21；
- 如果设置此滑轮关节的长度比为 2，那么当 length1=5 时，length2=26；
- 如果设置此滑轮关节的长度比为 3，那么当 length1=5 时，length2=31。

由此，可以更形象地看出“拉伸长度的比例”所表示的具体含义了，“拉伸长度的比例”就是表示两个滑轮到两个 Body 之间的两个长度会按照此比例关系成比例增长。

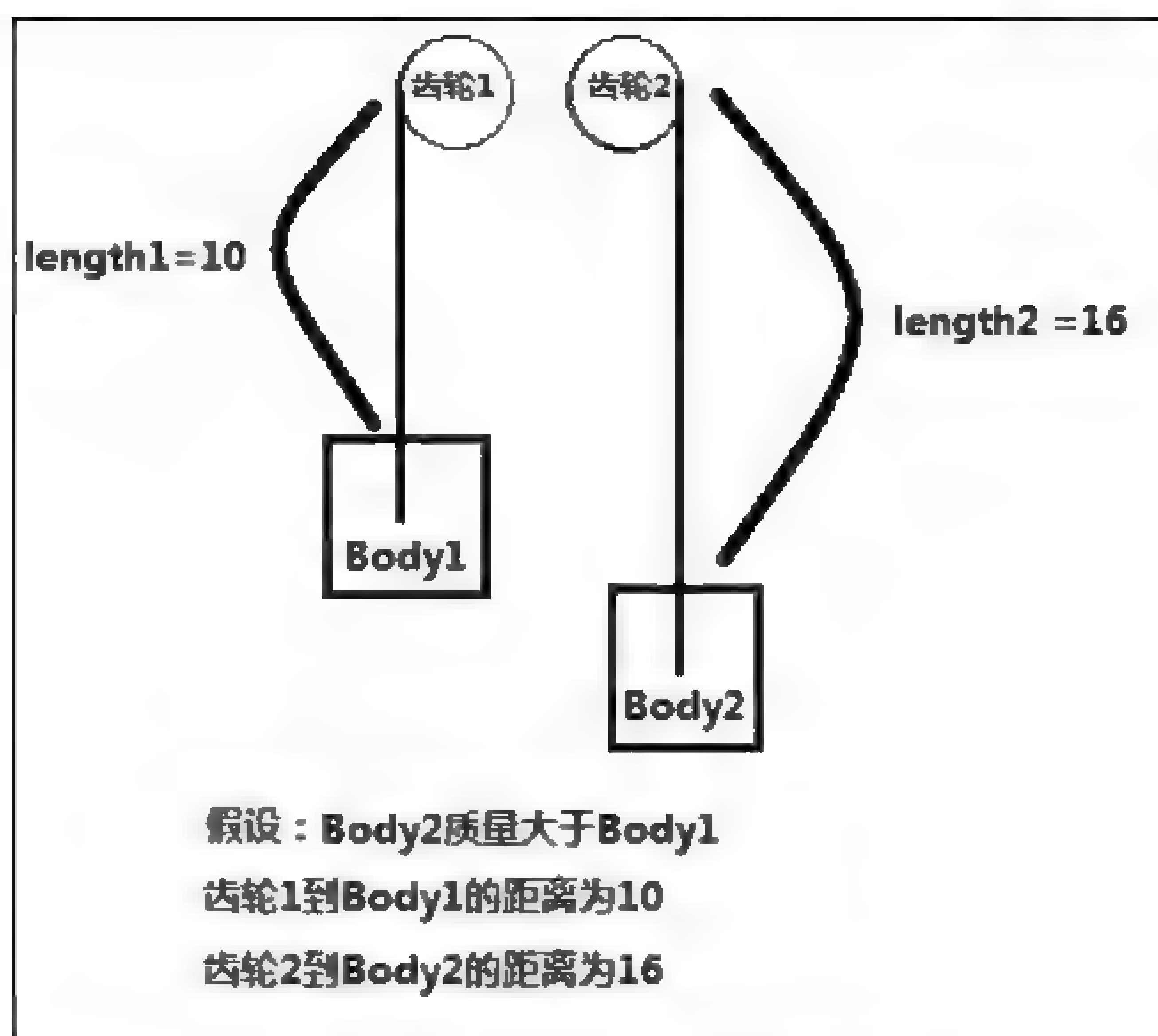


图 7-19 滑轮关节长度比演示图

滑轮关节的其他属性还包括：

- maxLength1：设置滑轮关节的第一条距离拉伸的最大长度；
- pd.maxLength2：设置滑轮关节的第二条距离拉伸的最大长度；
- ratio：设置滑轮关节的拉伸长度比例。

在使用滑轮关节时，唯一要提醒的一点是：设置滑轮拉伸的长度比例（ratio），一定要在滑轮关节数据初始化之后设置，否则此比例肯定会被滑轮关节数据初始化时的最后一个函数（设置拉伸长度比例）重新设置覆盖掉。

7.13.5 移动关节

移动关节（PrismaticJoint）起两个作用：一个作用是让物体沿着世界锚点进行移动，另外一个作用就是让绑定在移动关节上的两个 Body 进行相同的动作。

首先实现通过移动关节让 Body 移动，项目对应的源代码为“7-13-7-1（通过移动关节移动 Body）”。

添加一个 createPrismaticJointMove 函数，body1 的创建代码此处省略：

```
public PrismaticJoint createPrismaticJointMove() {
        //创建移动关节数据实例
        PrismaticJointDef pjd = new PrismaticJointDef();
        //初始化移动关节数据
        pjd.initialize(world.getGroundBody(), body1,
body1.getWorldCenter(), new Vec2(1, 0));
```

```
    //预设马达的最大力
    pjd.maxMotorForce = 10;
    //马达的最终力
    pjd.motorSpeed = 10;
    //启动马达
    pjd.enableMotor = true;
    //设置位移最小偏移值
    pjd.lowerTranslation = -80.0f / RATE;
    //设置位移最大偏移值
    pjd.upperTranslation = 80.0f / RATE;
    //启动限制
    pjd.enableLimit = true;
    //通过 world 创建一个移动关节
    PrismaticJoint pj = (PrismaticJoint) world.createJoint(pjd);
    return pj;
}
```

移动关节数据初始化函数为 initialize（Body groundBody, Body body1,Vec2 anchor , Vec2 axis），其参数说明如下：

- 第一个参数：传入接地 Body;
- 第二个参数：移动 Body 实例;
- 第三个参数：移动锚点;
- 第四个参数：物理世界坐标轴。

移动关节 Body 的移动方向取决于“马达的最终力（motorSpeed）”的值：

（1）当 motorSpeed>0 时，物理世界坐标轴（axis）的设置含义如下：

- new Vec2(1,0) 表示 Body 沿着 X 轴正方向进行移动;
- new Vec2 (-1,0) 表示 Body 沿着 X 轴反方向进行移动;
- new Vec2 (0,1) 表示 Body 沿着 Y 轴正方向进行移动;
- new Vec2 (0,-1) 表示 Body 沿着 Y 轴反方向进行移动。

（2）当 motorSpeed<0 时，物理世界坐标轴（axis）的设置含义如下：

- new Vec2 (1,0) 表示 Body 沿着 X 轴反方向进行移动;
- new Vec2 (-1,0) 表示 Body 沿着 X 轴正方向进行移动;
- new Vec2 (0,1) 表示 Body 沿着 Y 轴反方向进行移动;
- new Vec2 (0,-1) 表示 Body 沿着 Y 轴正方向进行移动。

在使用移动关节时需要注意如下两点：

- 在移动关节中“预设马达的最大力”无效，默认从 0 缓动;

- 绑定在移动关节上的Body在移动过程中不会发生角度的偏转。

项目运行效果如图7-20所示。

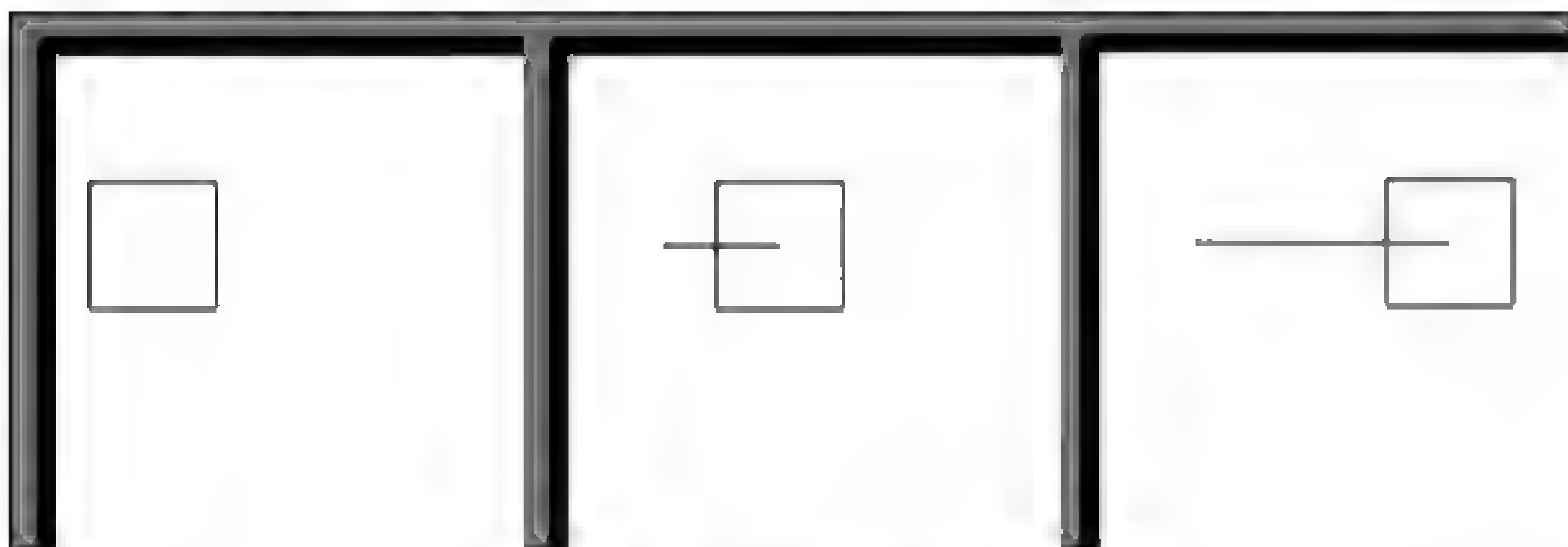


图7-20 移动关节-移动Body

下面来看一下如何通过移动关节实现让两个Body进行相同的动作，项目对应的源代码为“7-13-7-2（通过移动关节绑定两个Body动作）”。

添加一个createPrismaticJoint函数，此处省略两个Body的创建代码：

```
//创建移动关节
public PrismaticJoint createPrismaticJoint() {
      //创建移动关节数据实例
      PrismaticJointDef pjd = new PrismaticJointDef();
      //预设马达的最大力
      pjd.maxMotorForce = 10;
      //马达的最终力
      pjd.motorSpeed = 10;
      //启动马达
      pjd.enableMotor = true;
      //设置位移最小偏移值
      pjd.lowerTranslation = -3.0f / RATE;
      //设置位移最大偏移值
      pjd.upperTranslation = 3.0f / RATE;
      //启动限制
      pjd.enableLimit = true;
      //初始化移动关节数据
      pjd.initialize(body1, body2, body1.getWorldCenter(),
new Vec2(0, 1));
      //通过world创建一个移动关节
      PrismaticJoint pj = (PrismaticJoint) world.createJoint(pjd);
      return pj;
}
```

创建方式比较简单，唯一需要注意的就是移动关节数据初始化函数initialize，它的第一个参数与第二个参数分别是Body实例，而不使用接地body。

项目运行效果如图 7-21 所示。

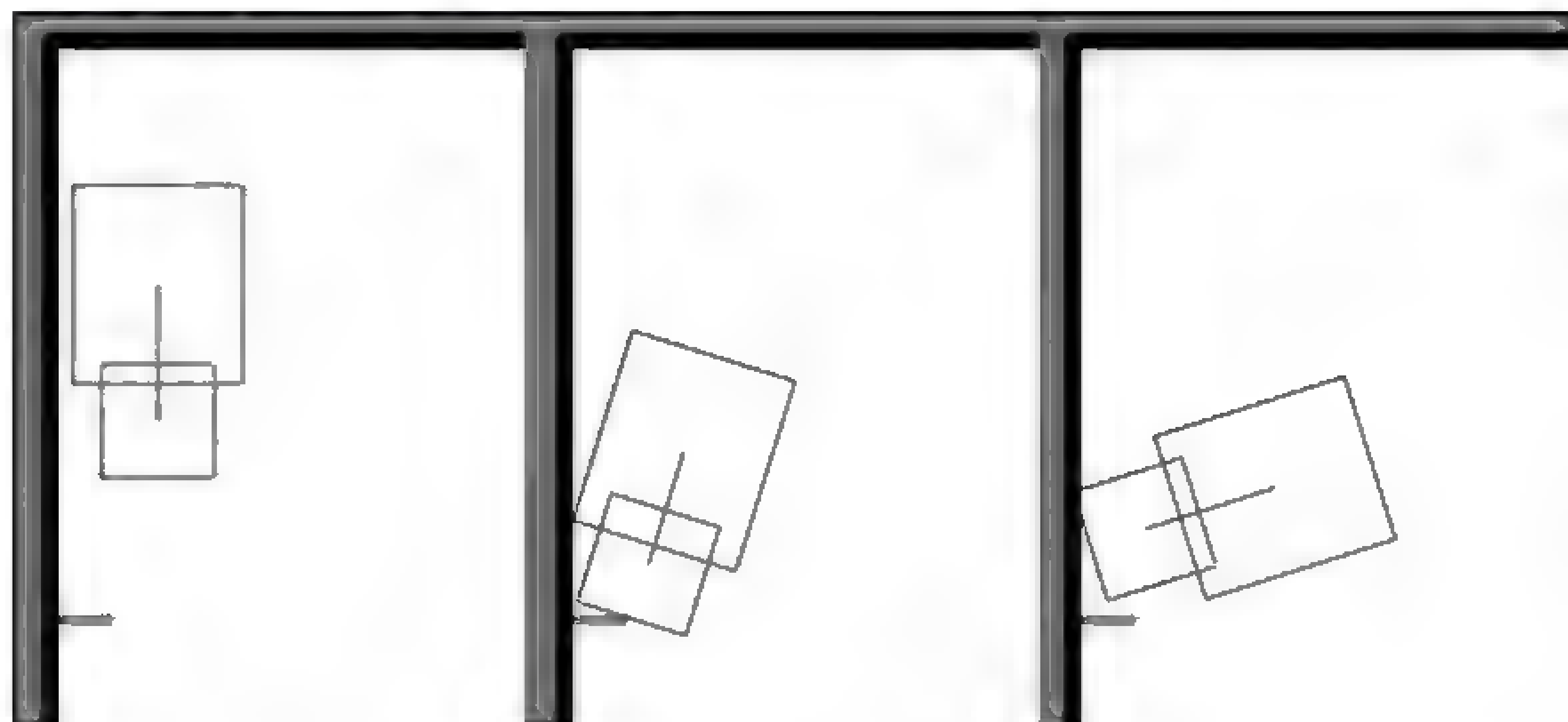


图 7-21　移动关节-绑定 Body 动作

通过图 7-21 可以很清晰的看到，移动关节绑定的两个 Body，一个 Body 发生旋转，另外一个则也会随之做出相同的旋转动作。利用移动关节的这些特点，然后加上旋转关节提供驱动力可以制作一个运动中的小车。

7.13.6　鼠标关节

鼠标关节（MouseJoint）指利用鼠标提供力的作用，拖拽 Body，Body 朝向鼠标点击的位置进行移动，其效果如同在 Body 与鼠标之间绑定了一个橡皮筋。

假设物理世界中存在一个 Body，然后点击屏幕产生一个鼠标点，那么 Body 会朝着如图 7-22 所示的箭头指向的方向运动。

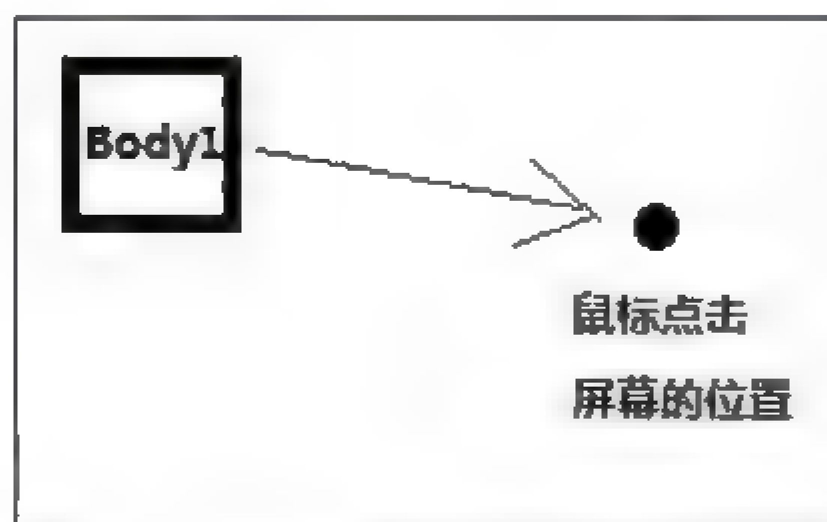


图 7-22　Body 向鼠标位置移动

下面来创建鼠标关节，并通过点击屏幕产生鼠标点以拖拽 Body，项目对应的源代码为“7-13-6（鼠标关节-拖拽 Body）”。首先创建鼠标关节（body1 的创建此处省略），在项目中添加一个 createMouseJoint 函数：

```
public MouseJoint createMouseJoint() {
      MouseJointDef mjd = new MouseJointDef();
      //设置鼠标关节的第一个Body实例（默认使用接地Body）
```

```
        mjd.body1 = world.getGroundBody();
        //设置鼠标关节的第二个 Body 实例(被拖拽的 Body)
        mjd.body2 = body1;
        //设置目标点的 X 坐标
        mjd.target.x = body1.getPosition().x;
        //设置目标点的 Y 坐标
        mjd.target.y = body1.getPosition().y;
        // body1.allowSleeping(false);//设置 body1 永不休眠
        // 设置鼠标关节的目标位置
        mjd.maxForce = 100;// 拉力
        //由 World 来创建鼠标关节
        MouseJoint mj = (MouseJoint) world.createJoint(mjd);
        return mj;
}
```

到这里代码只是创建了一个鼠标关节，并且将 Body 绑定在了鼠标关节中。接下来需要重写触屏函数，添加处理代码：

```
@Override
public boolean onTouchEvent(MotionEvent event) {
        //唤醒 Body1
        body1.wakeUp();
        mouseJoint.m_target.set(event.getX() / RATE, event.getY() / RATE);
        return true;
}
```

在上面代码中，首先唤醒 body1，因为对 body1 不产生操作时，body1 静止后会默认进入休眠状态，而且一旦当 body1 进入休眠，就无法响应鼠标关节的事件了；接下来设置鼠标关节的目标点，使之绑定在鼠标关节的 Body 向目标点运动。

运行项目效果如图 7-23 所示。

从图 7-23 可看出，鼠标关节往目标点进行移动时，会有一定的弹性，然后最终会趋向目标点并停止运动；这时，其弹性则可以通过鼠标关节来设置。

弹性度属性名为 m_gamma，此属性无法利用鼠标关节数据（MouseJointDef）来设置，只能通过鼠标关节实例（MouseJoint）来对其进行设置，方法如下。

```
mouseJoint.m_gamma = 1.0f;
```

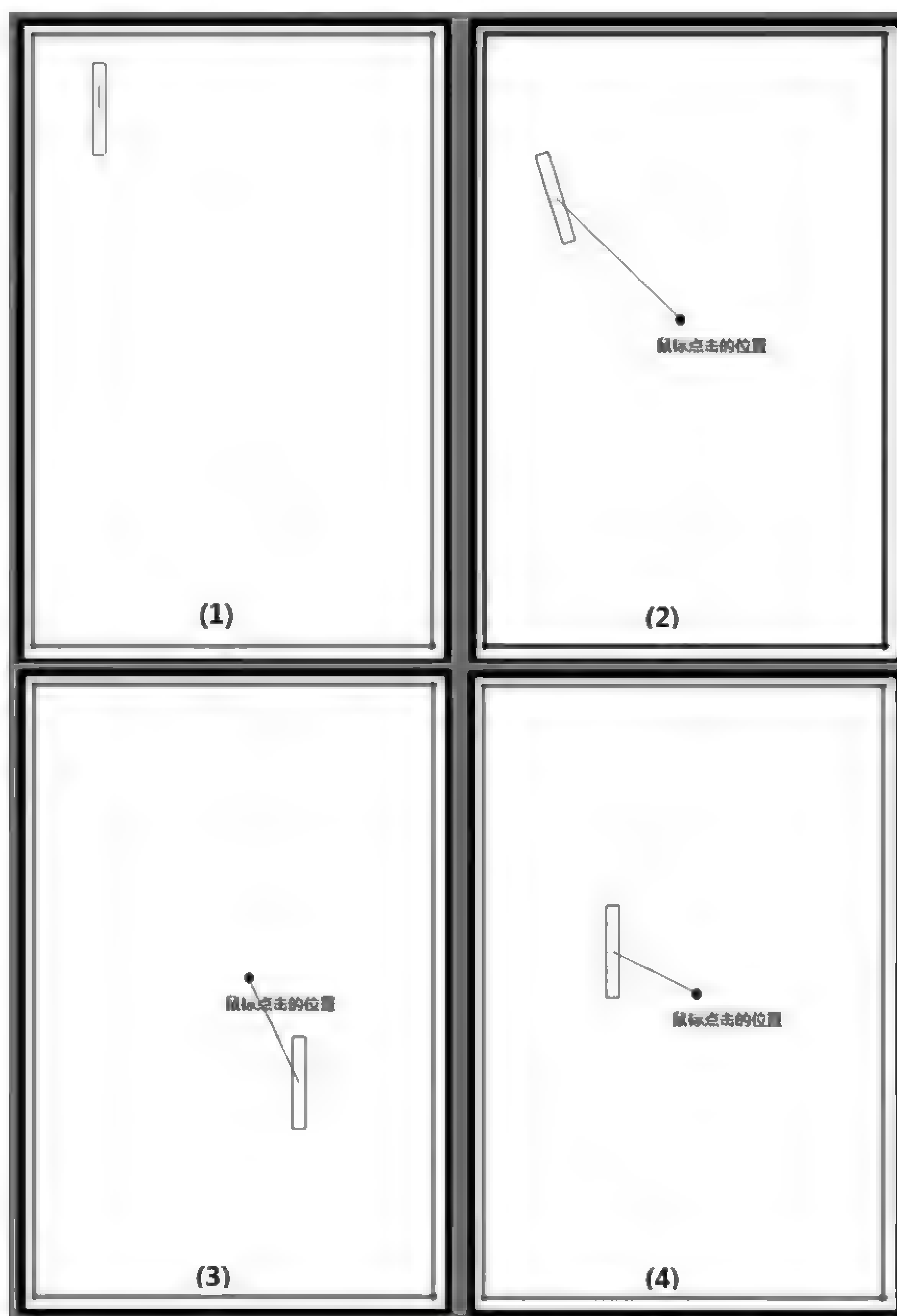


图 7-23　鼠标关节

7.14 通过 AABB 获取 Body

AABB 本身指的是一个区域范围，通过 AABB 的实例调用 lowerBound 与 upperBound 函

数来设置区域范围的大小。

在创建 World 物理世界时，传入一个 AABB 区域范围来表示物理世界的范围。当然它的作用不仅如此：在 World 类中，封装了 query 的方法，其参数中需传入一个 AABB 实例：

```
query(AABB aabb,int maxCount)
```

- 第一个参数：AABB 实例，区域范围；
- 第二个参数：最大重叠数。

query 方法的作用是返回在物理世界的 AABB 区域范围内存在的所有 Shape 实例，返回的最大值不可超过最大重叠数。

Shape 是皮肤的父类，它的子类有 CircleShape 与 PolygonShape，而且子类又派生了 CircleDef 与 PolygonDef 类。

我们封装了一个 getBodies 函数，示例项目对应的源代码为“7-14（AABB 获取 Body）”。

```
public Shape[] getBodies(float x, float y, float range, int maxCount) {
      AABB aabbBody = new AABB();
      aabbBody.lowerBound.set((x - range) / RATE, (y - range) / RATE);
      aabbBody.upperBound.set((x + range) / RATE, (y + range) / RATE);
      Shape[] shapes = world.query(aabbBody, maxCount);
      // 遍历此 aabb 范围中的 body，筛选操作
      for (int i = 0; i < shapes.length; i++) {
            if (shapes[i].getBody().isStatic()) {
                  // ...判定物体是否为静态
            }
            if (shapes[i].getBody().isSleeping()) {
                  // ...判定物体是否进入休眠
            }
      }
      return shapes;
}
```

在上面代码实现的 getBodies 函数中：

- 第一个参数：AABB 中心点 X 坐标；
- 第二个参数：AABB 中心点 Y 坐标；
- 第三个参数：AABB 的范围；
- 第四个参数：返回 AABB 区域中 Shape 最大重叠值。

获取 AABB 区域 Body 的步骤如下：

步骤1 首先创建一个 AABB 范围。

步骤2 通过 world.query()函数传入创建的 AABB 实例，并且设置最大重叠数值，得到所有 Shape 数组。

步骤3 有些情况我们并不想得到这个 AABB 范围内的所有 Body，那么可以通过遍历 Shape 取出所有 Body，利用筛选条件对其 Shape 数组进行筛选；在此封装的 getBodies 函数中筛选的条件有两个：

- 判定物体是否为静态物体；
- 物体是否进入休眠；

在本示例项目中，编者只是编写了判定条件的定义，并没有编写筛选出的物体的操作，大家可以根据需要写出适合的筛选条件，并进行处理。

项目运行效果如图 7-24 所示。

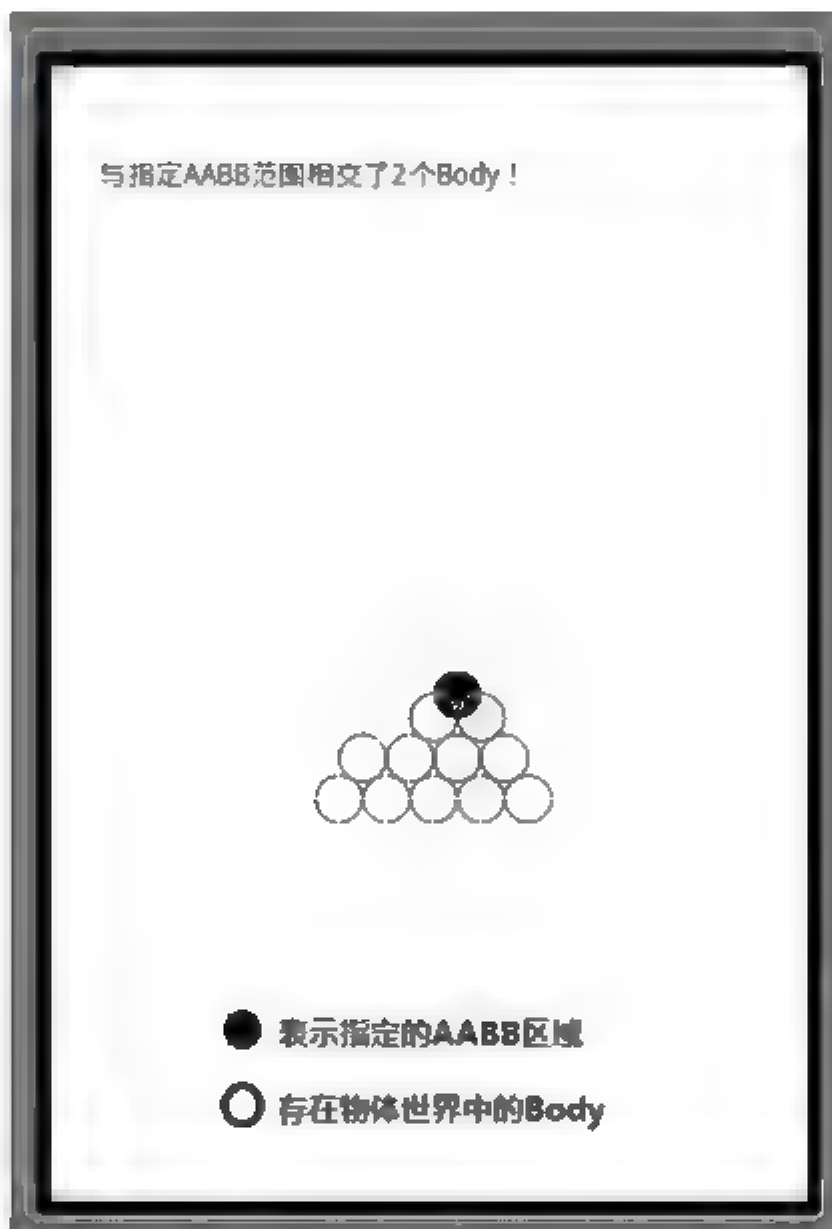


图 7-24　通过 AABB 获取其范围所有 Body

7.15 物体与关节的销毁

前面章节讲解了物体 Body 与关节 Joint 的创建，但是没有讲解如何销毁物体与关节；这里单独用一小节来强调销毁过程，也是为了提醒大家销毁操作的重要性。

想要摧毁一个 Body 或者 Joint，只要使用 World 类调用其 destory 函数即可，方法如下：

- 摧毁物体：world.destroyBody（Body body）；
- 摧毁关节：world.destroyJoint（Joint joint）。

代码很简单，但是千万不要认为摧毁一个物体和关节就这么简单。假如在 World 中存在

一个物体 Body，如果需要在按键事件中删除这个 Body，那么一般做法肯定是在按键事件中直接调用“world.destroyBody（Body body）;”语句，将需要删除的 Body 传入此函数，其实这种做法是错误的！出现问题的直接原因就是因为物理世界在模拟时出现异常。

大家都知道 world.step()函数是 World 物理世界进行模拟的函数，而且会将此函数一直放在线程中不断执行，让物理世界去不断地模拟。

也正是因为此函数所起的作用，使得删除一个 Body 或者删除一个 Joint 不当会造成物理世界模拟出现问题，抛出异常；异常的起因就是因为物理世界正在进行模拟，突然其中的 Body 被摧毁，造成这个 Body 所有的引用地方都成了野指针。

解决方案是：不管在程序的任何位置打算摧毁一个物体 Body 或者关节 Joint，其正确的方法是应该先将其存入一个容器中，然后在世界模拟完成之后，遍历容器，对容器中的 Body 或 Joint 进行摧毁操作。也就是说将摧毁 Body 或者摧毁 Joint 的操作放在世界完成模拟之后，对应摧毁的代码写在 world.step()函数之后。

7.16 本章小结

本章主要讲解了用于 2D 游戏开发的物理引擎——JBox2D 的使用方法，内容包括物理世界、矩形物体、多边形物体、圆形物体、关节、区域范围等对象的创建、使用和销毁方法。在下一章将讲述两个游戏实例说明如何使用 JBox2D 进行游戏开发。

第 8 章

Box2D 物理游戏实战

从本章节可以学习到：

- ❖ 迷宫小球游戏实战
- ❖ 堆房子游戏实战



8.1 迷宫小球游戏实战

首先我们看一下迷宫小球游戏项目的截图，游戏主界面与帮助界面如图 8-1 所示。游戏界面与游戏暂停界面如图 8-2 所示。

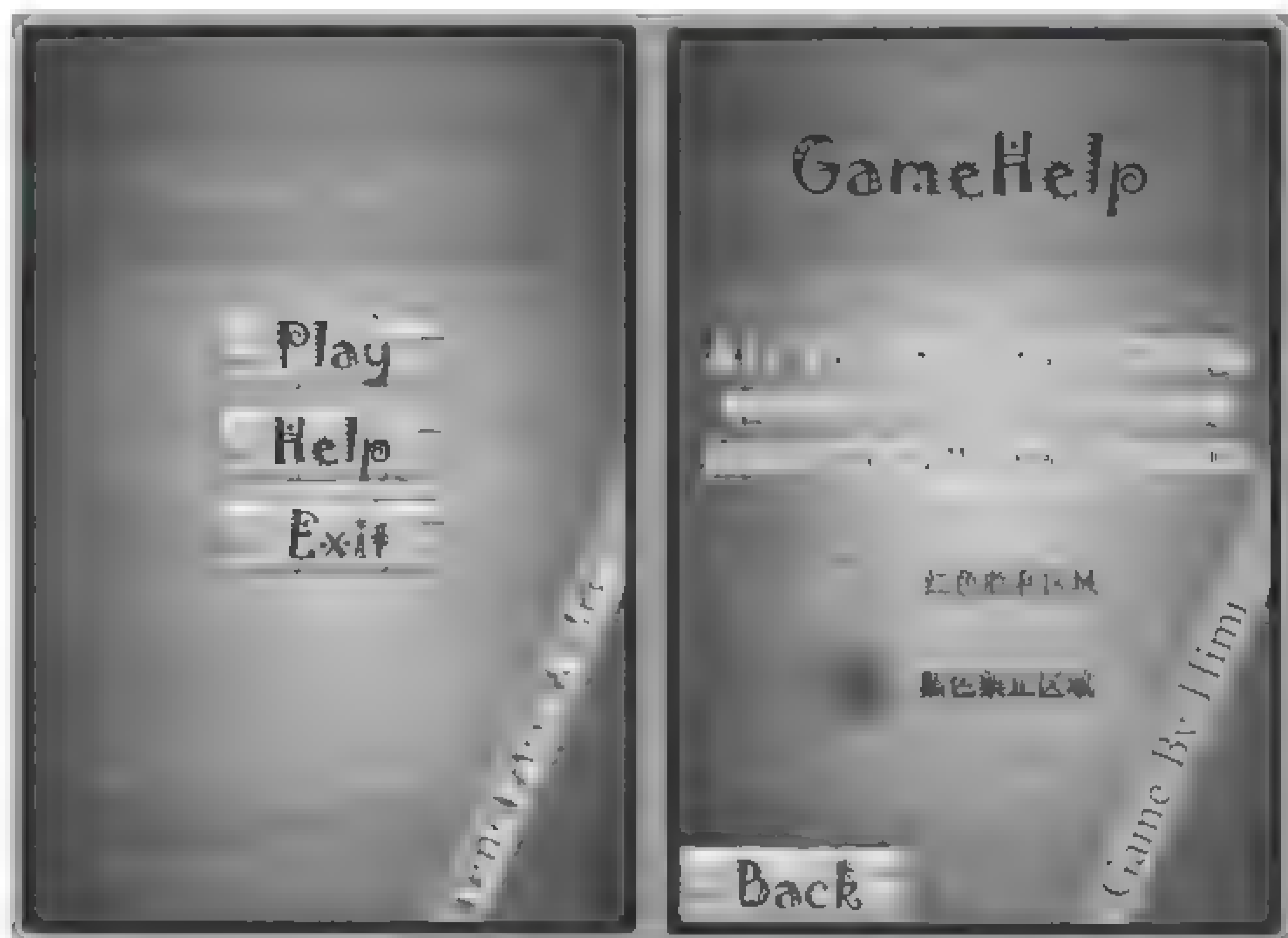


图 8-1　游戏主界面与帮助界面



图 8-2　游戏界面与游戏暂停界面

游戏胜利界面与失败界面如图 8-3 所示。



图 8-3　游戏界面胜利与失败界面

迷宫小球游戏项目简易流程如图 8-4 所示。

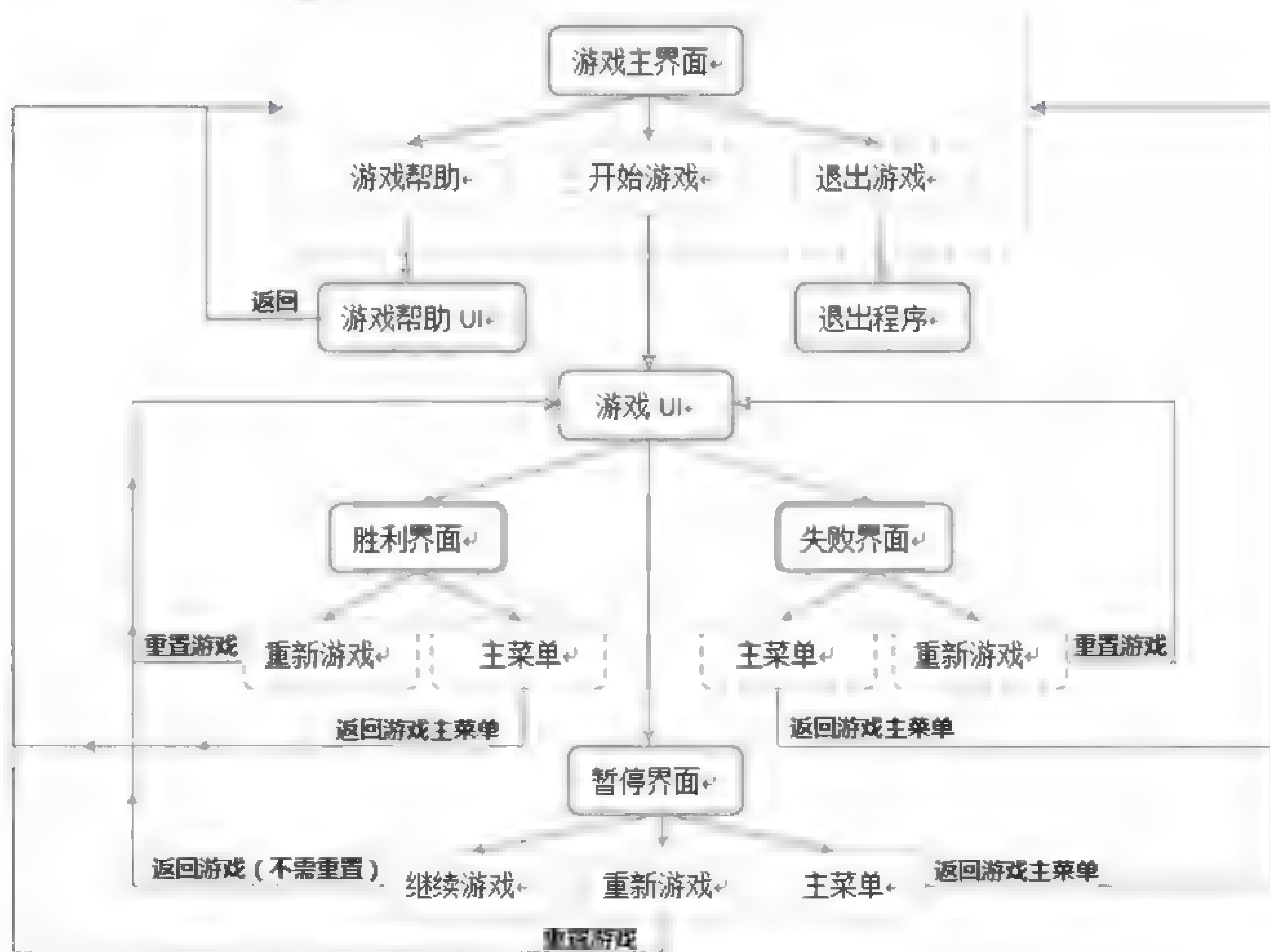


图 8-4　迷宫小球游戏项目简易流程

迷宫小球游戏玩法是：上下左右操作小球移动，小球碰触“黑色圆形区域”判定游戏失败；小球碰触“红色区域”判定游戏胜利。

下面我们对这个游戏项目进行分析。游戏本身没有难度，由几个简单的界面和一些按钮组成，主要操作是小球 Body 的移动。小球的移动其实可以通过改变物理世界的重力方向进行操控，只要想到这一点，游戏基本就成功一半了。

其实此款游戏可以结合 Android 手机的重力感应功能来做，让玩家通过转动手机屏幕来操控小球移动，这样会让整个游戏增色很多。但这里，作者没有结合重力感应来做的原因有两点：

- 便于演示与讲解，因为此游戏实例重点在于讲解 Box2D 部分为主；
- 照顾还没有真机的开发者，因为模拟器无法模拟手机的重力感应。

首先新建项目“BallGame”，如图 8-5 所示，然后将 Jbox2d 引擎的 JAR 包添加到项目中，项目对应的源代码为“8-1（迷宫小球）”。



图 8-5　新建项目“BallGame”

模拟器的配置如图 8-6 所示：



图 8-6 模拟器的配置

在新建项目中添加资源文件，如图 8-7 所示：

drawable-mdpi

序列	图片说明	图片名称	图片缩略图	序列	图片说明	图片名称	图片缩略图
1.	游戏背景	game_bg.png		12.	游戏暂停背景	smallbg.png	
2.	主菜单背景	menu_bg.png		13.	游戏失败背景	gamelost.png	
3.	帮助背景	helpbg.png		14.	胜利背景	gamewin.png	
4.	障碍物(横)	sh.png		15.	障碍物(竖)	ss.png	
5.	边界(横)	h.png		16.	边界(竖)	s.png	
6.	失败物体	lostbody.png		17.	胜利物体	winbody.png	
7.	主角小球	ball.png		18.	应用图标	icon.png	
8.	选项Menu	menu_menu.png	Menu	19.	选项Help	menu_help.png	Help
9.	选项Play	menu_play.png	Play	20.	选项Resume	menu_resume.png	Resume
10.	选项Back	menu_back.png	Back	21.	选项Replay	menu_replay.png	Replay
11.	选项Exit	menu_exit.png	Exit				

图 8-7 在新建项目中添加的资源文件

修改主类 MainActivity:

```
public class MainActivity extends Activity {
	//声明一个主类（便于使用）
	public static MainActivity main;
	@Override
	public void onCreate(Bundle savedInstanceState) {
		super.onCreate(savedInstanceState);
		//实例本类
		main = this;
		//设置全屏
	getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
				WindowManager.LayoutParams.FLAG_FULLSCREEN);
		//去除应用程序标题
		this.requestWindowFeature(Window.FEATURE_NO_TITLE);
		//设置竖屏
setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_PORTRAIT);
		//显示自定义MySurfaceView实例
		setContentView(new MySurfaceView(this));
	}
	//添加一个程序退出方法
	public void exit() {
		//退出应用程序
		System.exit(0);
	}
}
```

添加自定义的 MySurfaceView 类，游戏使用 SurfaceView 框架，并且创建 Box2D 物理世界。

```
public class MySurfaceView extends SurfaceView implements Callback,
Runnable {
	private Thread th;
	private SurfaceHolder sfh;
	private Canvas canvas;
	private Paint paint;
	private boolean flag;
	// ----添加一个物理世界---->>
	// 屏幕映射到现实世界的比例 30px：1m;
	private final float RATE = 30;
	// 声明一个物理世界对象
	private World world;
	// 声明一个物理世界的范围对象
	private AABB aabb;
	// 声明一个重力向量对象
	private Vec2 gravity;
	// 物理世界模拟的的频率
	private float timeStep = 1f / 60f;
```

```
    // 迭代值，迭代越大模拟越精确，但性能越低
    private int iterations = 10;
    public MySurfaceView(Context context) {
            super(context);
            //保持屏幕常亮
            this.setKeepScreenOn(true);
            //获取句柄实例
            sfh = this.getHolder();
            //添加本类回调函数
            sfh.addCallback(this);
            //实例画笔
            paint = new Paint();
            //设置画笔无锯齿
            paint.setAntiAlias(true);
            //设置画笔样式为空心
            paint.setStyle(Style.STROKE);
            //设置按键焦点
            this.setFocusable(true);
            //设置触屏焦点
            this.setFocusableInTouchMode(true);
            // --添加一个物理世界--->>
            // 实例化物理世界的范围对象
            aabb = new AABB();
            // 实例化物理世界重力向量对象
            gravity = new Vec2(0, 10);
            // 设置物理世界范围的左上角坐标
            aabb.lowerBound.set(-100, -100);
            // 设置物理世界范围的右下角坐标
            aabb.upperBound.set(100, 100);
            // 实例化物理世界对象
            world = new World(aabb, gravity, true);
    }
    public void surfaceCreated(SurfaceHolder holder) {
            //线程循环标识位
            flag = true;
            //线程实例化
            th = new Thread(this);
            //启动线程
            th.start();
    }
    public void myDraw() {
            try {
                    //获取画布
                    canvas = sfh.lockCanvas();
                    //黑色刷屏
                    canvas.drawColor(Color.BLACK);
```

```
            } catch (Exception e) {
                Log.e("Himi", "myDraw is Error!");
            } finally {
                if (canvas != null)
                    sfh.unlockCanvasAndPost(canvas);
            }
        }
        /**
         * 触屏按键事件处理
         */
        @Override
        public boolean onTouchEvent(MotionEvent event) {
            return true;
        }
        /**
         * 实体键盘按键处理
         */
        @Override
        public boolean onKeyDown(int keyCode, KeyEvent event) {
            return super.onKeyDown(keyCode, event);
        }
        /**
         *逻辑处理
         */
        public void Logic() {
            // --开始模拟物理世界--->>
            world.step(timeStep, iterations);// 物理世界进行模拟
        }
        public void run() {
            while (flag) {
                myDraw();
                Logic();
                try {
                    Thread.sleep((long) timeStep * 1000);
                } catch (Exception ex) {
                    Log.e("Himi", "Thread is Error!");
                }
            }
        }
        public void surfaceChanged(SurfaceHolder holder, int format, int
width, int height) {
        }
        public void surfaceDestroyed(SurfaceHolder holder) {
            flag = false;
        }
}
```

从 8.1 节的项目截图中，可以看出游戏使用了很多菜单选项，所以需要封装一个菜单选项类。

新建一个按钮类 HButton。按钮应该拥有“坐标”与“宽高”属性，“绘制函数”以及“是否被点击函数”。通过项目截图可看出每个按钮都显示个图片，那么按钮还应该有个“图片”的基本属性。按钮类 Hbutton 代码如下：

```
public class HButton {
	//按钮xy坐标与宽高
	private int x, y, w, h;
	//按钮资源图片
	private Bitmap bmp;
	/**
	 * 按钮构造函数
	 * @param bmp 按钮图片资源
	 * @param x   按钮X坐标
	 * @param y   按钮Y坐标
	 */
	public HButton(Bitmap bmp, int x, int y) {
		this.x = x;
		this.y = y;
		this.bmp = bmp;
		this.w = bmp.getWidth();
		this.h = bmp.getHeight();
	}
	/**
	 * 绘制按钮
	 * @param canvas  画布实例
	 * @param paint   画笔实例
	 */
	public void draw(Canvas canvas, Paint paint) {
		canvas.drawBitmap(bmp, x, y, paint);
	}
	/**
	 * 判定按钮是否被点击
	 * @param event 触屏事件参数
	 */
	public boolean isPressed(MotionEvent event) {
		//判定用户是否点击屏幕
		if (event.getAction() == MotionEvent.ACTION_DOWN) {
			//通过按钮坐标宽高与触屏的坐标进行判定按钮是否被点击
			if (event.getX() <= x + w && event.getX() >= x) {
				if (event.getY() <= y + h && event.getY() >= y) {
					return true;
				}
			}
```

```
            }
            //没有点击返回 false
            return false;
        }
        //获取按钮的宽
        public int getW() {
            return w;
        }
        //获取按钮的高
        public int getH() {
            return h;
        }
        //获取按钮的 X 坐标
        public int getX() {
            return x;
        }
        //设置按钮的 X 坐标
        public void setX(int x) {
            this.x = x;
        }
        //获取按钮的 Y 坐标
        public int getY() {
            return y;
        }
        //设置按钮的 Y 坐标
        public void setY(int y) {
            this.y = y;
        }
}
```

因为这是个 Box2D 物理游戏，所以 Body 的种类从项目效果图中明显看出有两种，一种圆形 Body，一种矩形 Body。为了便于 Body 与图形传递运动数据，我们应该对应设计两个类，一个 Circle 类，一个 Rect 类。

首先创建一个圆形类 MyCircle：

```
public class MyCircle {
        // 圆形的宽高与半径
        float x, y, r, angle;
        //圆形 Body 图片资源
        private Bitmap bmp;
        /**
         * 圆形图片构造函数
         * @param bmp 资源图片
         * @param x  圆形图形 X 坐标
         * @param y  圆形图形 Y 坐标
         * @param r  圆形图形半径
```

```
	 */
	public MyCircle(Bitmap bmp, float x, float y, float r) {
		this.bmp = bmp;
		this.x = x;
		this.y = y;
		this.r = r;
	}
	/**
	 * 圆形图形绘制函数
	 * @param canvas 画布实例
	 * @param paint    画笔实例
	 */
	public void drawArc(Canvas canvas, Paint paint) {
		canvas.save();
		canvas.rotate(angle, x + r, y + r);
		canvas.drawBitmap(bmp, x, y, paint);
		canvas.restore();
	}
	// 设置圆形图形的 X 坐标
	public void setX(float x) {
		this.x = x;
	}
	//设置圆形图形的 Y 坐标
	public void setY(float y) {
		this.y = y;
	}
	//设置圆形图形的角度
	public void setAngle(float angle) {
		this.angle = angle;
	}

	// 获得圆形图形的半径
	public float getR() {
		return r;
	}
}
```

然后创建一个矩形类 MyRect:

```
public class MyRect {
	// 矩形图形的图片
	private Bitmap bmp;
	// 矩形图形的坐标
	private float x, y;
	/**
	 * 矩形图片构造函数
	 * @param bmp 资源图片
```

```
	 * @param x  矩形图形X坐标
	 * @param y  矩形图形Y坐标
	 * @param r  矩形图形半径
	 */
	public MyRect(Bitmap bmp, float x, float y) {
		this.bmp = bmp;
		this.x = x;
		this.y = y;
	}
	/**
	 * 矩形图形绘制函数
	 * @param canvas 画布实例
	 * @param paint     画笔实例
	 */
	public void drawMyRect(Canvas canvas, Paint paint) {
		canvas.drawBitmap(bmp, x, y, paint);
	}
	// 设置矩形图形的X坐标
	public void setX(float x) {
		this.x = x;
	}
	//设置矩形图形的Y坐标
	public void setY(float y) {
		this.y = y;
	}
	//获取矩形图形的宽
	public int getW() {
		return bmp.getWidth();
	}
	//获取矩形图形的高
	public int getH() {
		return bmp.getHeight();
	}
}
```

按钮与两个Body图形类设计完毕，接下来我们开始分析整体游戏框架。从图8-4所示的游戏流程图中，可以看出整个游戏主要可分为以下3个状态：

- 状态一：主菜单；
- 状态二：帮助界面；
- 状态三：正在进行游戏。

正在进行游戏的状态又分为以下3种情况：

- 暂停游戏；
- 游戏胜利；

- 游戏失败。

以上游戏状态可以在主类MySurfaceView中定义，代码如下：

```
// 声明游戏状态常量
//主菜单界面
private final int GAMESTATE_MENU = 0;
// 帮助界面
private final int GAMESTATE_HELP = 1;
//游戏界面
private final int GAMESTATE_GAMEING = 2;
//当前游戏状态为主菜单
private int gameState = GAMESTATE_MENU;
```

定义游戏状态之后，在逻辑、绘制、触屏与实体按键等函数中也根据戏状态来处理：

```
switch (gameState) {
      case GAMESTATE_MENU:
            break;
      case GAMESTATE_HELP:
            break;
      case GAMESTATE_GAMEING:
            break;
}
```

这样一来游戏整体的结构就很清晰了，根据不同的游戏状态写出对应的处理代码与绘制方法。

回到主视图类，在MySurfaceView中，添加创建圆形Body与矩形Body两个函数（为了便于讲解，图片按钮等资源的声明与实例在此省略）：

```
//在物理世界中添加矩形Body
public Body createRect(Bitmap bmp, float x, float y, float w, float h,
float density) {
      PolygonDef pd = new PolygonDef();
      pd.density = density;
      pd.friction = 0.8f;
      pd.restitution = 0.3f;
      pd.setAsBox(w / 2 / RATE, h / 2 / RATE);
      BodyDef bd = new BodyDef();
      bd.position.set((x + w / 2) / RATE, (y + h / 2) / RATE);
      Body body = world.createBody(bd);
      body.m_userData = new MyRect(bmp, x, y);
      body.createShape(pd);
      body.setMassFromShapes();
      return body;
}
```

```
//在物理世界中添加圆形Body
public Body createCircle(Bitmap bmp, float x, float y, float r, float
density) {
        CircleDef cd = new CircleDef();
        cd.density = density;
        cd.friction = 0.8f;
        cd.restitution = 0.3f;
        cd.radius = r / RATE;
        BodyDef bd = new BodyDef();
        bd.position.set((x + r) / RATE, (y + r) / RATE);
        Body body = world.createBody(bd);
        body.m_userData = new MyCircle(bmp, x, y, r);
        body.createShape(cd);
        body.setMassFromShapes();
        body.allowSleeping(false);
        return body;
}
```

创建Body过程就不再详述，主要提醒一点：

在创建圆形Body时，设置了body.allowSleeping（false）这个属性，此属性设置Body永远不进入睡眠。设置其属性的作用是：防止主角小球进入睡眠，导致改变物理世界的重力方向无法移动已休眠的Body。

防止小球进入睡眠的方法除了此种方法外，也可以在每次改变重力方向时，使用"bodyBall.wakeUp();"语句来唤醒小球也可解决。

接下来完成添加主菜单的UI。主菜单界面比较简单，包括一张背景图片与三个按钮，添加代码如下：

```
public void myDraw() {//绘制函数
        ...
        switch (gameState) {
        case GAMESTATE_MENU:
                //绘制主菜单背景
                canvas.drawBitmap(bmp_menubg, 0, 0, paint);
                //绘制 Play 按钮
                hbPlay.draw(canvas, paint);
                //绘制 Help 按钮
                hbHelp.draw(canvas, paint);
                //绘制 Exit 按钮
                hbExit.draw(canvas, paint);
                break;
        case GAMESTATE_HELP:
                break;
        case GAMESTATE_GAMEING:
                break;
        }
```

```
        ...
}
@Override//触屏按键监听
public boolean onTouchEvent(MotionEvent event) {
        ...
        switch (gameState) {
        case GAMESTATE_MENU:
                //判断 Play 按钮是否被点击
                if (hbPlay.isPressed(event)) {
                        //Play 按钮被点击开始游戏
                        gameState = GAMESTATE_GAMEING;
                //判断 Help 按钮是否被点击
                } else if (hbHelp.isPressed(event)) {
                        //Play 按钮被点击进入游戏游戏帮助界面
                        gameState = GAMESTATE_HELP;
                //判断 Exit 按钮是否被点击
                } else if (hbExit.isPressed(event)) {
                        //Exit 按钮被点击调用退出函数
                        MainActivity.main.exit();
                }
                break;
        case GAMESTATE_HELP:
                break;
        case GAMESTATE_GAMEING:
                break;
        }
        ...
        return true;
}
```

添加帮助界面。帮助界面包括一张背景图片和一个返回按钮，添加代码如下：

```
public void myDraw() {//绘制函数
        ...
        switch (gameState) {
        case GAMESTATE_MENU:
                ...
                break;
      case GAMESTATE_HELP:
                //绘制帮助界面背景
                canvas.drawBitmap(bmp_helpbg, 0, 0, paint);
                //绘制 Back 按钮
                hbBack.draw(canvas, paint);
               break;
        case GAMESTATE_GAMEING:
               break;
        }
```

```
        ...
}
@Override//触屏按键监听
public boolean onTouchEvent(MotionEvent event) {
        ...
        switch (gameState) {
        case GAMESTATE_MENU:
            ...
            break;
        case GAMESTATE_HELP:
              //判断 Back 按钮是否被点击
              if (hbBack.isPressed(event)) {
                     //Back 按钮被点击进入主菜单界面
                     gameState = GAMESTATE_MENU;
              }
              break;
        case GAMESTATE_GAMEING:
              break;
        }
        ...
        return true;
}
```

添加游戏界面。相对于帮助与主菜单界面而言，游戏 UI 比较复杂，除了背景与主角、障碍物等绘制，还要处理在游戏时满足胜利与失败条件的事件，以及暂停事件。

当游戏进行时，一旦满足游戏暂停、胜利、失败条件，必须暂停游戏逻辑，让物理世界停止模拟，所以还要添加逻辑函数的处理代码：

```
public void myDraw() {//绘制函数
        ...
        switch (gameState) {
        case GAMESTATE_MENU:
              ...
              break;
        case GAMESTATE_HELP:
              ...
              break;
        case GAMESTATE_GAMEING:
              //绘制游戏背景
              canvas.drawBitmap(bmp_gamebg, 0, 0, paint);
              //遍历物理世界中所有存在的 Body;
              Body body = world.getBodyList();
              for (int i = 1; i < world.getBodyCount(); i++) {
                     if ((body.m_userData) instanceof MyRect) {
                            MyRect rect = (MyRect) (body.m_userData);
                            rect.drawMyRect(canvas, paint);
```

```
                } else if ((body.m_userData) instanceof MyCircle) {
                        MyCircle mcc = (MyCircle)
            (body.m_userData);
                        mcc.drawArc(canvas, paint);
                }
                body = body.m_next;
            }
            //当游戏暂停或失败或成功时
            if (gameIsPause || gameIsLost || gameIsWin) {
                // 当游戏暂停或失败或成功时画一个半透明黑色矩形，突出界面
                Paint paintB = new Paint();
                paintB.setAlpha(0x77);
                canvas.drawRect(0, 0, screenW, screenH, paintB);
            }
            // 游戏暂停
            if (gameIsPause) {
                //绘制暂停背景
                canvas.drawBitmap(bmp_smallbg, screenW / 2 -
bmp_smallbg.getWidth() / 2, screenH / 2 - bmp_smallbg.getHeight() / 2,
paint);
                //绘制 Resume 按钮
                hbResume.draw(canvas, paint);
                //绘制 Replay 按钮
                hbReplay.draw(canvas, paint);
                //绘制 Menu 按钮
                hbMenu.draw(canvas, paint);
            } else
            //游戏失败
            if (gameIsLost) {
                //绘制游戏背景
                canvas.drawBitmap(bmpLostbg, screenW / 2 -
bmpLostbg.getWidth() / 2, screenH / 2 - bmpLostbg.getHeight() / 2, paint);
                //绘制 Replay 按钮
                hbReplay.draw(canvas, paint);
                //绘制 Menu 按钮
                hbMenu.draw(canvas, paint);
            } else
            //游戏胜利
            if (gameIsWin) {
                //绘制失败背景
                canvas.drawBitmap(bmpWinbg, screenW / 2 -
bmpWinbg.getWidth() / 2, screenH / 2 - bmpWinbg.getHeight() / 2, paint);
                //绘制 Replay 按钮
                hbReplay.draw(canvas, paint);
                //绘制 Menu 按钮
                hbMenu.draw(canvas, paint);
```

```
            }
            break;
        }
        ...
}
@Override//触屏按键监听
public boolean onTouchEvent(MotionEvent event) {
        ...
        switch (gameState) {
        case GAMESTATE_MENU:
              ...
              break;
        case GAMESTATE_HELP:
              break;
              ...
        case GAMESTATE_GAMEING:
        // 游戏暂停、失败、胜利
        //(因为游戏失败和游戏胜利中按钮处理事件一样，所以就不需要重复写)
              if (gameIsPause || gameIsLost || gameIsWin) {
                    if (hbResume.isPressed(event)) {
                          gameIsPause = false;
                    } else if (hbReplay.isPressed(event)) {
              //因为在重置前小球可能拥有力，所以重置游戏要先使用 putToSleep()方法
              //让其 Body 进入睡眠，并让 Body 停止模拟，速度置为 0
                          bodyBall.putToSleep();
                          // 然后对小球的坐标进行重置
                          bodyBall.setXForm(new Vec2((bmpH.getHeight() +
bmpBall.getWidth() / 2 + 2) / RATE, (bmpH.getHeight() + bmpBall.getWidth()
/ 2 + 2) / RATE),  0);
                          //并且设置默认重力方向为向下
                          world.setGravity(new Vec2(0, 10));
                          //唤醒小球
                          bodyBall.wakeUp();
                          //游戏暂停、胜利、失败条件还原默认 false
                          gameIsPause = false;
                          gameIsLost = false;
                          gameIsWin = false;
                    } else if (hbMenu.isPressed(event)) {
                          bodyBall.putToSleep();
                          // 然后对小球的坐标进行重置
                          bodyBall.setXForm(new Vec2((bmpH.getHeight() +
bmpBall.getWidth() / 2 + 2) / RATE, (bmpH.getHeight() + bmpBall.getWidth()
/ 2 + 2) / RATE), 0);
                          //并且设置默认重力方向为向下
                          world.setGravity(new Vec2(0, 10));
                          //唤醒小球
```

```
                            bodyBall.wakeUp();
                            //重置游戏状态为主菜单
                            gameState = GAMESTATE_MENU;
                            //游戏暂停、胜利、失败条件还原默认 false
                            gameIsPause = false;
                            gameIsLost = false;
                            gameIsWin = false;
                        }
                 }
                break;
        }
        ...
        return true;
}
//游戏逻辑函数
public void Logic() {
        switch (gameState) {
        case GAMESTATE_MENU:
                break;
        case GAMESTATE_HELP:
                break;
        case GAMESTATE_GAMEING:
                // 游戏没有满足暂停、失败、胜利条件时
                if (!gameIsPause && !gameIsLost && !gameIsWin) {
                        // --开始模拟物理世界--->>
                        world.step(timeStep, iterations);
                        Body body = world.getBodyList();
                        for (int i = 1; i < world.getBodyCount(); i++) {
                                if ((body.m_userData) instanceof MyRect) {
                                  MyRect rect = (MyRect) (body.m_userData);
                                  rect.setX(body.getPosition().x * RATE -
rect.getW() / 2);
                                  rect.setY(body.getPosition().y * RATE -
rect.getH() / 2);
                                } else if ((body.m_userData) instanceof MyCircle)
{
                                  MyCircle mcc = (MyCircle) (body.m_userData);
                                  mcc.setX(body.getPosition().x * RATE -
mcc.getR());
                                  mcc.setY(body.getPosition().y * RATE -
mcc.getR());
                                  mcc.setAngle((float) (body.getAngle() * 180 /
Math.PI));
                                }
                                body = body.m_next;
                        }
```

```
            }
            break;
        }
}
```

三个游戏状态绘制与按键处理都添加完毕之后，开始添加主角小球的控制移动代码：

```
@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
      // 当前游戏状态不处于正在游戏中时，屏蔽“返回”实体按键,避免程序进入后台;
      if (keyCode == KeyEvent.KEYCODE_BACK && gameState !=
GAMESTATE_GAMEING) {
            return true;
      }
      switch (gameState) {
      case GAMESTATE_MENU:
            break;
      case GAMESTATE_HELP:
            break;
      case GAMESTATE_GAMEING:
            // 游戏没有暂停、失败、胜利
            if (!gameIsPause && !gameIsLost && !gameIsWin) {
                  //如果方向键左键被按下
                  if (keyCode == KeyEvent.KEYCODE_DPAD_LEFT)
                        //设置物理世界的重力方向向左
                        world.setGravity(new Vec2(-10, 2));
                  //如果方向键右键被按下
                  else if (keyCode == KeyEvent.KEYCODE_DPAD_RIGHT)
                        //设置物理世界的重力方向向右
                        world.setGravity(new Vec2(10, 2));
                  //如果方向键上键被按下
                  else if (keyCode == KeyEvent.KEYCODE_DPAD_UP)
                        //设置物理世界的重力方向向上
                        world.setGravity(new Vec2(0, -10));
                  //如果方向键下键被按下
                  else if (keyCode == KeyEvent.KEYCODE_DPAD_DOWN)
                        //设置物理世界的重力方向向下
                        world.setGravity(new Vec2(0, 10));
                  //如果返回键被按下
                  else if (keyCode == KeyEvent.KEYCODE_BACK) {
                        //进入游戏暂停界面
                        gameIsPause = !gameIsPause;                  }
            }
            //屏蔽“返回”实体按键,避免程序进入后台;
            return true;
      }
      return super.onKeyDown(keyCode, event);
```

```
}
```

在方向键按下设置物理世界重力方向的时候，向左和向右的 Y 轴的向量值并没有设置为零，这是因为 Box2D 物理世界是理想化的，一旦设为零，小球向左或向右时，就会有种很飘、很假的感觉，当然这些也是作者的个人感觉，大家可以根据实际情况进行调整。

虽然在实体按键中，对“返回”按钮进行了处理和屏蔽，但是还要考虑 Home 按键的处理，虽然点击 Home 不会像点击 Back 按键一样重启当前 Activity，但是也会导致执行 SurfaceView 中的 surfaceCreated 函数，所以如果在 surfaceCreated 中添加了一些初始化的代码，那么可以如下处理：

```
//SurfaceView 创建
public void surfaceCreated(SurfaceHolder holder) {
      //防止 Home 按键导致游戏重置
      if (gameState == GAMESTATE_MENU) {
          ...
      }
      ...
}
```

在此游戏中，游戏分为三种状态。那么，通过 gameState 这个记录当前游戏状态的变量，可以有效防止玩家在点击 Home 按键再次回到游戏时，又重新加载初始化 surfaceCreated 函数中的代码。

8.2 堆房子游戏实战

在上一节中，详细介绍了一个 Box2D 小游戏，通过对这个游戏的分析，一方面为了让大家加深对游戏制作流程的理解，另一方面也学习了 Body 的创建与操作方法。

本节将再继续为大家讲解一个“堆房子”的游戏，这个游戏只写了 Demo，没有完成游戏胜利、失败，以及其他界面。学习此 Demo 的实现可以让大家掌握两个知识点：

- 通过关节与 Body 相结合制作游戏；
- 动态创建 Body。

至于游戏未完成的部分，大家可以自由发挥将其完成，这里就不再说明了。

堆房子游戏玩法是：触摸屏幕使悬挂屏幕上方的方块掉落。游戏的运行效果如图 8-8 所示。

首先新建项目“BuildHouse”，然后将 Jbox2D 引擎 JAR 包添加到项目中，如图 8-9 所示，项目对应的源代码为“8-2（堆房子）”。

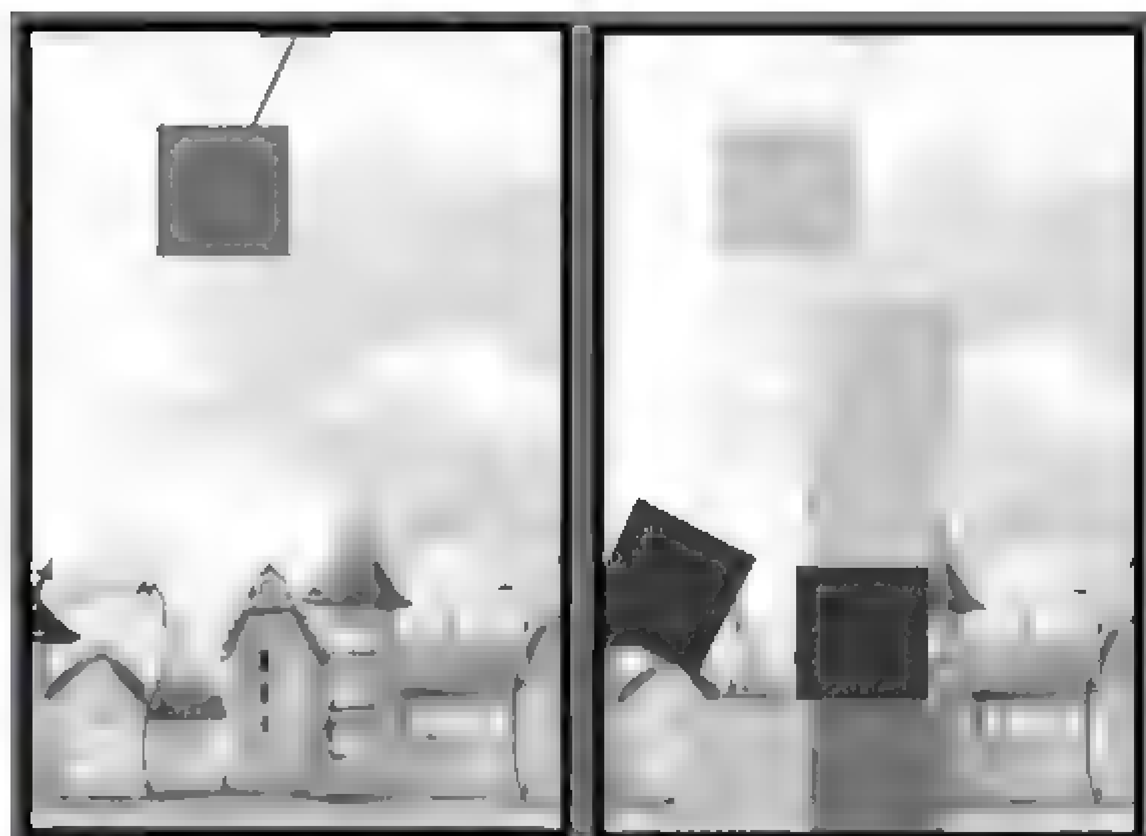


图 8-8　堆房子游戏截图

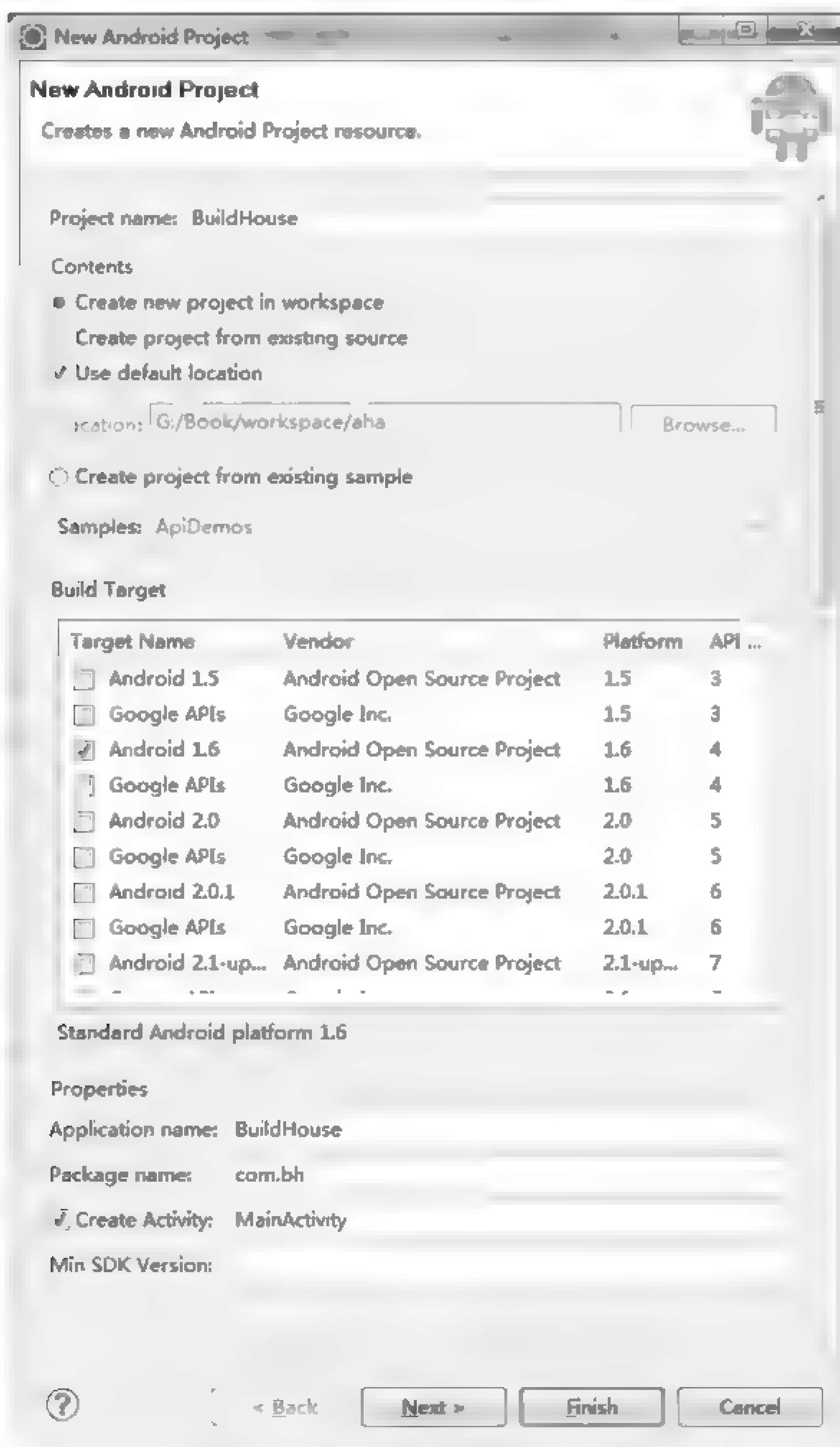


图 8-9　新建项目“BuildHouse”

模拟器的配置如图 8-10 所示：



图 8-10　模拟器的配置

在项目中添加资源文件如图 8-11 所示：

drawable-mdpi

序列	图片说明	图片名称	图片
1.	游戏背景	background.jpg	
2.	砖块1	tile1.png	
3.	砖块2	tile2.png	
4.	砖块3	tile3.png	

图 8-11　在项目中添加资源文件

首先在“MainActivity”类中设置游戏全屏，并且显示自定义的 View——“MySurfaceView”类。MySurfaceView 游戏框架中有基本的绘制、逻辑、线程等函数。这两个类在前文已经多次介绍过，大家再熟悉不过了，这里不再详述。

从图 8-8 所示的项目效果中可以看出，游戏大致可分为两种 Body，一种砖块 Body，一种正常矩形 Body。那我们就先添加这两个对应的 Body 图形类 MyRect 和 MyTile。

MyRect 类的代码如下:

```
public class MyRect {
	//矩形图形的位置、宽高与角度
	private float x, y, w, h, angle;
	//矩形图形的初始化
	public MyRect(float x, float y, float w, float h) {
		this.x = x;
		this.y = y;
		this.w = w;
		this.h = h;
	}
	//矩形图形绘制函数
	public void drawRect(Canvas canvas, Paint paint) {
		canvas.save();
		canvas.rotate(angle, x + w / 2, y + h / 2);
		canvas.drawRect(x, y, x + w, y + h, paint);
		canvas.restore();
	}
	//设置矩形图形的 X 坐标
	public void setX(float x) {
		this.x = x;
	}
	//设置矩形图形的 Y 坐标
	public void setY(float y) {
		this.y = y;
	}
	//设置矩形图形的角度
	public void setAngle(float angle) {
		this.angle = angle;
	}
	//获取矩形图形的宽
	public float getWidth() {
		return w;
	}
	//获取矩形图形的高
	public float getHeight() {
		return h;
	}
}
```

MyTile 类的代码如下:

```
public class MyTile {
```

```
//砖块的坐标、宽高与角度
private float x, y, w, h, angle;
//砖块的图片资源
private Bitmap bmp;
//砖块初始化
public MyTile(float x, float y, float w, float h, Bitmap bmp) {
        this.x = x;
        this.y = y;
        this.w = w;
        this.h = h;
        this.bmp = bmp;
}
//砖块的绘制
public void drawMyTile(Canvas canvas, Paint paint) {
        canvas.save();
        canvas.rotate(angle, x + w / 2, y + h / 2);
        canvas.drawBitmap(bmp, x, y, paint);
        canvas.restore();
}
//获取砖块的X坐标
public float getX() {
        return x;
}
//获取砖块的Y坐标
public float getY() {
        return y;
}
//设置砖块的x坐标
public void setX(float x) {
        this.x = x;
}
//设置砖块的Y坐标
public void setY(float y) {
        this.y = y;
}
//获取砖块的宽
public float getWidth() {
        return w;
}
//获取砖块的高
public float getHeight() {
        return h;
}
//设置砖块角度
public void setAngle(float angle) {
        this.angle = angle;
```

```
    }
}
```

两个类结构比较简单，而且实现方法基本相同，主要的区别在于 MyTile 比 MyRect 类中多了图片属性。接下来下面实现一下“MySurfaceView”类。

首先初始化图片资源、创建物理世界，这里就不再详述。在代码中添加 3 个重要函数：

（1）创建一般的矩形 Body 函数：

```
//创建矩形 Body
public Body createPolygon(float x, float y, float w, float h, float angle,
float density) {
        PolygonDef pd = new PolygonDef();
        pd.density = density;
        pd.friction = 0.8f;
        pd.restitution = 0.3f;
        pd.setAsBox(w / 2 / RATE, h / 2 / RATE);
        BodyDef bd = new BodyDef();
        bd.position.set((x + w / 2) / RATE, (y + h / 2) / RATE);
        bd.angle = (float) (angle * Math.PI / 180);
        Body body = world.createBody(bd);
        body.m_userData = new MyRect(x, y, w, h);
        body.createShape(pd);
        body.setMassFromShapes();
        return body;
}
```

（2）创建砖块 Body 函数：

```
//创建装块 Body
public Body createMyTile(float x, float y, float w, float h, float angle,
            float density) {
        PolygonDef pd = new PolygonDef();
        pd.density = density;
        pd.friction = 0.8f;
        pd.restitution = 0.3f;
        pd.setAsBox(w / 2 / RATE, h / 2 / RATE);
        pd.filter.groupIndex = 2;
        BodyDef bd = new BodyDef();
        bd.position.set((x + w / 2) / RATE, (y + h / 2) / RATE);
        bd.angle = (float) (angle * Math.PI / 180);
        Body body = world.createBody(bd);
        //实例随机库
        random = new Random();
        //取出 0-2 的随机整数
        ran = random.nextInt(3);
        //bmp ：绑定在距离关节上的动态 Body
```

```
//bmpTile1、bmpTile2、bmpTile3 分别是三种砖块的图片资源
      if (ran == 0)
            bmp = bmpTile1;
      else if (ran == 1)
            bmp = bmpTile2;
      else if (ran == 2)
            bmp = bmpTile3;
      body.m_userData = new MyTile(x, y, w, h, bmp);
      body.createShape(pd);
      body.setMassFromShapes();
      return body;
}
```

创建步骤与一般矩形 Body 创建类似，只是赋值 m_userData 属性时，为了让每次随机创建不同，所以通过 Rondom 获取随机数，通过随机数来对应不同的图片资源，从而创建不同图片的 Body。

（3）创建距离关节函数：

```
//创建距离关节
public DistanceJoint createDistanceJoint(Body body1, Body body2) {
      //创建距离关节信息
      DistanceJointDef dj = new DistanceJointDef();
      dj.body1 = body1;
      dj.body2 = body2;
      //初始化距离关节
      dj.initialize(body1, body2, body1.getWorldCenter(),
body2.getWorldCenter());
      //通过World创建一个距离关节对象
      return (DistanceJoint) world.createJoint(dj);
}
```

这些预备工作做好之后，开始往视图中添加游戏元素。因为方块 Body 的运动轨迹成摇摆状，所以可以利用距离关节来实现，只要固定一个距离关节中的一个 Body 为静态，另外一个 Body 为动态即可。添加游戏元素的步骤如下：

步骤1 添加一个静态 Body，放置在屏幕最下方，用于放置掉落的 Body 出屏。

```
createPolygon(0, getHeight(), getWidth() - 10, 10, 0, 0);
```

步骤2 添加距离关节。

```
//实例游戏初始两个绑定距离关节的Body
//bodyWall : 固定在屏幕顶端的静态Body
bodyWall = createPolygon(bodyWallX, bodyWallY, bodyWallW, bodyWallH,
0, 0);
//bodyHouse: 砖块Body
```

```
bodyHouse = createMyTile(bodyWallX / 2, bodyWallY + bodyWallH + 50,
bmpTile1.getWidth(), bmpTile1.getHeight(), 0, 1);
//dj:距离关节对象
// ---添加一个距离关节
dj = createDistanceJoint(bodyWall, bodyHouse);
```

游戏中，方块 bodyWall 为距离关节中的静态物体，bodyHouse 为距离关节中的动态物体，所以我们将 bodyWall 的质量设为 0。

步骤3 启动物理世界模拟、Body 与图形数据传值以及绘制。

逻辑函数 Logic 是线程不断调用的函数，代码如下:

```
//逻辑处理函数
private void logic() {
      //物理世界模拟
      world.step(stepTime, iteraTions);
      //遍历 Body，进行 Body 与图形之间的传递数据
      Body body = world.getBodyList();
      for (int i = 1; i < world.getBodyCount(); i++) {
             //判定 m_userData 中的数据是否为 MyRect 实例
             if ((body.m_userData) instanceof MyRect) {
                    MyRect rect = (MyRect) (body.m_userData);
                    rect.setX(body.getPosition().x * RATE -
rect.getWidth() / 2);
                    rect.setY(body.getPosition().y * RATE -
rect.getHeight() / 2);
                    rect.setAngle((float) (body.getAngle() *
180 / Math.PI));
             } else if ((body.m_userData) instanceof MyTile) {
                    //判定 m_userData 中的数据是否为 MyTile 实例
                    MyTile tile = (MyTile) (body.m_userData);
                    tile.setX(body.getPosition().x * RATE -
tile.getWidth() / 2);
                    tile.setY(body.getPosition().y * RATE -
tile.getHeight() / 2);
                    tile.setAngle((float) (body.getAngle() *
180 / Math.PI));
             }
             body = body.m_next;
      }
}
```

绘制函数 MyDraw 也是需要线程不断调用的函数，代码如下：

```
//绘制函数
```

```
public void draw() {
       ...
       //获取画布实例
       canvas = sfh.lockCanvas();
       //刷屏
       canvas.drawColor(Color.BLACK);
       //绘制游戏背景图
       canvas.drawBitmap(bmpBg, 0, -Math.abs(getHeight() -
bmpBg.getHeight()), paint);
       //遍历绘制Body
       Body body = world.getBodyList();
       for (int i = 1; i < world.getBodyCount(); i++) {
              if ((body.m_userData) instanceof MyRect) {
                     MyRect rect = (MyRect) (body.m_userData);
                     rect.drawRect(canvas, paint);
              } else if ((body.m_userData) instanceof MyTile) {
                     MyTile tile = (MyTile) (body.m_userData);
                     tile.drawMyTile(canvas, paint);
              }
              body = body.m_next;
       }
       if (bodyWall != null && bodyHouse != null) {
              if (dj != null) {
                     //设置画笔颜色
                     //lineColor:int 数组 ，保存三种颜色值;
                     //分别表示不同方砖的悬挂绳颜色
                     paint.setColor(lineColor[ran]);
                     //设置画笔的粗细程度
                     paint.setStrokeWidth(3);
                     //绘制悬挂绳
                     canvas.drawLine(bodyWall.getPosition().x * RATE,
bodyWall.getPosition().y * RATE, bodyTemp.getPosition().x * RATE,
bodyTemp.getPosition().y * RATE, paint);
                     }
              }
       ...
}
```

绘制距离关节，其实就是连接距离关节中的两个Body所在物理世界的中心点。

步骤4 接下来处理触屏事件。

游戏中触屏，让摇摆的方块下落，做法一般有两种：

- 触屏时，摧毁距离关节，绑定在距离关节中的自由落体，下次创建新的Body绑定新的距离关节中；

- 触屏时，创建一个新的 Body，其位置等属性应与绑定在距离关节上动态 Body 一致，让其自由落体。

这里我们使用第一种方式，首先创建两个布尔值和一个临时 Body，然后再实现触屏事件和逻辑处理函数：

```
private Body bodyTemp;//永远指向距离关节的动态 Body 实例
private boolean isDown;// 表示是否动态 Body 是否开始下落
private boolean isDeleteJoint;// 是否删除距离关节
//触屏事件
@Override
public boolean onTouchEvent(MotionEvent event) {
	...
	if (isDown == false) {
		if (event.getAction() == MotionEvent.ACTION_DOWN) {
			//删除关节
			isDeleteJoint = true;
			//动态 Body 下落
			isDown = true;
		}
	}
	...
	return true;
}
```

关节的删除操作并没有在触屏中直接处理，其原因之前也详细说过，放置物理世界模拟时，出现野指针，导致报错。所以触屏中我们用 isDeleteJoint 布尔值来标识该删除关节，而删除操作放在世界模拟之后处理。

```
//逻辑处理函数
private void logic() {
	...
	//动态 Body 是否下落
	if (isDown == true) {
		//计时器计时
		isDownCount++;
		//计时器时间
		if (isDownCount % 120 == 0) {
			//创建新的动态 Body-砖块，并且使用临时 Body 保存最新动态 Body
			bodyTemp = createMyTile(bodyWallX / 2, bodyWallY + 
bodyWallH + 50, bmpTile1.getWidth(), bmpTile1.getHeight(), 0, 1);
			//创建新的距离关节
			dj = createDistanceJoint(bodyWall, bodyTemp);
			//计时器重置
			isDownCount = 0;
			//下落标识重置
```

```
                        isDown = false;
                }
        }
        ...
}
```

整个游戏流程很简单，实现起来也不难。这里，还是要再次提醒大家，Body 物体与 Joint 关节的删除操作一定要放置在物理世界模拟后操作，这是使用 Box2d 开发游戏最值得注意的地方。

8.3 本章小结

本章详细剖析了迷宫小球和堆房子游戏的实现方法。从迷宫小球游戏实战中，读者可以加深对游戏制作流程的理解，同时巩固了 Body 创建与操作的方法。从堆房子游戏实践中，读者可以掌握通过关节与 Body 相结合制作游戏的方法。